MASTER IN COMPUTER SCIENCE

A tag-based approach to software product line implementation

Gauthier, Christophe

Award date:
2010

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur

Faculté d'informatique - Faculty of Computer Science
2009-2010

A Tag-based Approach to Software Product Line Implementation

Christophe Gauthier

Master Thesis in Computer Science, 2009-2010 at FUNDP, Namur Belgium

A thesis presented to the Faculty of Computer Science in partial fulfillment of the requirements for the Degree Master of Computer Science.

Mémoire présenté en vue de l'obtention du grade de master et/ou licencié en informatique.

A Tag-based Approach to Software Product Line Implementation

Master Thesis in Computer Science, 2009-2010 at FUNDP, Namur Belgium

Author: Christophe GAUTHIER

Supervisor: Patrick HEYMANS

Co-Supervisor: Quentin BOUCHER

Arnaud HUBAUX

Acknowledgment

Thanks to everyone who made this work possible. Prof Patrick Heymans and his assistants for their help in developing XToF, writing and correcting this thesis. Prof Anne-Margaret Storey, the members of the CHISEL team for welcoming me into their team and for their support, especially Del Meyers, the developer of TagSEA for his help in Eclipse development, Supreetee Saddul for correcting my english and my parents for everything else.

Summary

Developing a set of similar softwares separately is time and money consuming. Software Product Line (SPL) aims at developing the set of software as a whole. It uses features as an abstraction for functions that can be selected or not. Compositional approaches imply changes in paradigm and language to implement SPL; annotative approaches do not. Compositional approaches implement features in distinct modules that are composed to generate a specific product. Annotative approaches use annotations to register which portion of source code are associated to a feature, and using this information can generate products. Annotative approaches provide a lightweight approach to implementing SPL. However, compositional approaches provide better feature traceability and safety. Beyond the different annotative approaches, the tagging approach provides a tool independent and syntactically safe approach while remaining a lightweight approach. It uses tags of features to associate source code with features.

The tagging approach is tool independent, but the disadvantages when contrasted to the comparative approaches can be reduced by using a tool to support the approach. The tool's main requirements are: the integration with a Feature Model (FM), the traceability function, pruning, configuration (the selection of features) and the program understanding for SPL. The tool called XToF uses an architecture made of plugins, is extensible to new languages and provides many functions to support the approach. Among them, visualization at source code and project levels enables the developer to understand how features are implemented.

Résumé

Pour développer un ensemble de logiciels qui répondent à des besoins similaires, les techniques d'ingénierie du logiciel classique ne sont pas adaptées. Il est nécessaire de développer chaque logiciel séparément. Cette manière de procéder est consommatrice de ressources. L'ingénierie en famille de produits logiciels (SPL) propose des techniques et méthodes pour développer un ensemble de logiciels similaire comme un tout. Les fonctions qui constituent les différents produits sont regroupés en fonctionnalités (*features*) qui peuvent être sélectionnées ou non pour générer un produit spécifique. Les approches par composition de cette ingénierie imposent un changement de paradigme et de langage, et imposent un type de structure pour la conception du logiciel. Dans ces approches, l'implémentation de chaque fonctionnalité est réalisée dans des modules séparés qui sont composés pour former un produit. Ces approches sont dites *lourdes*. A l'opposé, les approches par annotation utilise des annotations pour associer des régions de code source à des fonctionnalités. Ce sont des approches *légères*. Elles ont cependant des défauts par rapport aux approches par composition qui rendent la traçabilité de l'implémentation des fonctionnalités obscure et sont sources d'erreurs syntaxiques. Parmi les différentes approches par annotation, l'approche par tags utilise les noms des fonctionnalités comme mots-clés pour associer des régions du code source à celles-ci.

L'approche par tags est indépendante d'un outil spécifique et n'est pas source d'erreur syntaxique lors de la génération de produits. Cependant, un support réalisé par un outil serait utile pour fournir la traçabilité à l'implémentation des fonctionnalités. Les différents moyens de supports fournis par l'outil sont l'élagage du code (*pruning*), l'intégration avec un modèle des fonctionnalités (*feature model*), la configuration de produits (sélection des fonctionnalités), traçabilité des fonctionnalités et compréhension de programme adapté aux SPL. L'outil développé pour répondre à ces exigences est appelé XToF, il possède une architecture constituée de plugins, peut-être étendu pour gérer des langages supplémentaires. Une partie de la recherche consiste en des visualisations pour comprendre l'implémentation des fonctionnalités à deux niveaux d'abstraction, au niveau du code source et du projet.

Appendix

1. Introduction

Software is widely used, companies using software are faced with the question of how to obtain a software that fits its particular needs. As each company may not be able to develop their own solution, they may buy existing ones. Then the software developers are faced with the issue of how to adapt a standard software for each user's specific needs. In addition, each software, tough having some differences, also have common parts that should be developed only once to reduce the time and money needed.

Software engineering is well suited for building a software having methods and theories to support the developer's work. However, it is not adapted for developing a set of similar softwares. For that reason, Software Product Line (SPL) engineering was created. SPL aims at providing a support to the developer interested in developing a set of similar softwares. For example, a software could be designed as an SPL. Instead of having only one software and the users that have to adapt to it, it is possible to produce a software specific for each user. Each of these software would address different needs and therefore propose different functions. However, each product, being softwares of the same set, would also have some similarity and should share functions. SPL proposes to develop the whole set of software together instead of developing each software of the set separately. It helps reduce the cost and time needed.

SPL engineering helps model the set of software by using an abstraction for functions: features. A feature can be selected and present in a product or absent, unselected. Selecting some features and not selecting others enable to develop a specific software called a product. SPL engineering also provides methods and techniques to develop and implement the set of software (every possible product for a given SPL).

However, most implementations techniques impose heavy changes. They may require changes to paradigms and/or languages, or impose a specific structure for the design and therefore, cannot be easily adopted without significantly changing the existing implementation process. This work intends to propose a method to implement software product lines in a light way. A light way should not impose strong requirements on the developer and should be close to the software engineering he is familiar with. Of course, the design of the SPL in itself is a new mandatory task. For this reason, it cannot be removed and is not part of this work. The design of the SPL comprises the modelling of the features and their constraints (some features may be incompatible or require the use of other features). When implementing SPL, the developer should also be able to understand how the features are implemented, as he may not be working alone on the SPL. To reduce the weight of this task, this work also intends to support the developer in understanding feature implementation, also known as program understanding for SPL. This can be done by using visualizations techniques.

This works focuses on the implementation of the SPL. It analyses different methods used to implement SPL, presents and criticizes the lightest approach – the tagging approach which uses tags to annotate features in source code without requiring any other changes. To reduce the weight of the approach and some of its disadvantages, a tool can support the approach. This will be described through its requirements, design and a guided tour.

This work is separated into two parts, the background and the contribution. The first provides informations about existing methods to implement and visualize SPL. Chapter two describes several notions for SPL, for example what is a SPL, how is it modelled and the two main approaches to implement them, the compositional and the annotative approach. Chapter three describes and compares the main annotative approaches used to implement SPL. The adopted tagging approach is also described in this chapter. Chapter 4 presents visualization techniques adapted for visualization of features in source code and projects. The second parts contains the chapters 5 to 10. Chapter 5 compares the tagging approach to the other annotative approaches and describes the requirements for a tool support. Chapter 6 explains the architecture of the tool. Chapter 7 is a guided tour of the tool and of its support mechanisms. Chapter 8 illustrates the approach and the tool. Future work is described in chapter 9. This works concludes in chapter 10.

Part 1. Background

This part provides the reader with a survey of Software Product Line implementation approaches. It then analyses existing methods adapted to the lightweight goal. Finally visualization techniques for features in source code and projects are reviewed.

2. Software Product Lines

In this chapter, the notion of software product lines will first be introduced through examples. A description of problem will be then given which software product lines are supposed to address. The software product line engineering with its advantages will also be described and the following section will provide an insight of how to develop for several needs through the use of variability, itself explained in an other part. After this general discussion, the text will focus on the process which is the subject of this works, the implementation of software product lines. It will describe and compare the two main approaches used to implement the software product. The next chapter will describe more precisely the approach chosen in the last section of this chapter.

2.1. What are Software Product Lines?

Today, software is of increasing importance. The number of places where software is used is large, from mobile phones to cars. The former are becoming more complex and offer increasingly more functions. Some have even transformed into mobile gaming stations. While software brings new functions to existing products, such as fridges that can automatically order groceries. They known what foodstuffs are inside and propose a list for order.

Usually, there is more than one single product that will receive software. The manufacturers have several models. So, the developers must develop as many software. Each software will be specific to each model. Manufacturers produce several models because they address different needs. Developing each product separately costs time and money. For each kind of product, for example a family of mobile phones, there is a set of softwares to develop. The softwares will have some differences as each model is different. However, they have common functions. Every phone can be used to make phone calls.

Software Product Line (SPL) engineering is aimed at developing software in this context: one specific software for each need. It provides a way to realize development in this context but also to reduce its time and cost. SPL is a paradigm created to take the different functions, also called features, into account in the development and reduce the time and money needed. It reduces the amount of work by using the commonality between the products and helps manage the variability and the complexity of the products. Software product line engineering is aimed at developing multiple similar software systems rather than a few single systems. SPL engineering adds new processes to the regular software engineering.

Software product line is a '*software-intensive system that shares a common, managed set of features satisfying a particular market segment's specific needs or mission and that are developed from a common set of core assets in a prescribed way'* [29].

The idea is to develop a general software and customize it for each need. It reduces the development of a set of softwares to a single general one. SPL engineering takes into account the differences between the softwares to provide the customization. For example, a developer wants to develop an enterprise resource planning (ERP) software to sell. He will have several customers, but they each have different needs. Some only buy and sell products, while others also manufacture products. The developer can use the SPL engineering to develop a set of software that will be adapted to each customer. The general software could be a complete software that can do everything, from managing purchases and sales to managing production. The developer customizes the software and only keeps functions that the client needs. This can be done in several ways, for example, by compiling or dynamically loading only the required functions.

2.1.1. Examples of Success Stories

Software product lines is used in many companies manufacturing products, such as HP, Nokia, Ericsson and Boeing. The best example is the mobile phone. For example,

Nokia uses SPL engineering to manufacture its products. Another example is HP and its printers. Although these two examples are referring to embedded softwares, they are not the only application of software product lines. Software systems composed only of software can also benefit from software product line engineering.

Mobile phones have become increasingly present. To sell to more people, manufacturers have to produce more than one product. Each buyer may not have the same interest in a phone; some want a good camera, others want to listen to music, etc. Models differs in their hardware and their functions and therefore the software that runs on them has to be specific to each model.

One solution to have a software adapted to each product is to redevelop it each time. However, it would cost much more time and money to proceed this way, especially when companies are interested in reducing costs to make profits. This solution would also have problems for example more bugs would be present in the numerous software programs produced by the company and every process of the software development cycle would have to be repeated several times. This is a time and money consuming solution.

A better solution is to improve re-use. Developers have for a long time tried to reduce the amount of code that needs to be written. They used routines, objects, etc. These mechanisms provided only limited re-use. In this example, the mobiles have one function common to all of them: making phone calls. Planning and developing it only once, would reduce time and costs. Some mobile phones also have a camera function. Not every phone has one, but factorizing its development could still reduce the work that needs to be done. This is taking into account the commonality[1] of the systems to reduce the amount of software that needs to be developed.

The previous example placed emphasis on the significance of knowing the commonality in software engineering. However, this knowledge must also take into account the specificity of each product. Printers were cited as an other example. There are two main printer technologies: the inkjet and the laser. Even if every printer can print, they do not all use the same process. As there are only two types of printers, re-using the software enables economies. In this example, the number of possibilities are limited. However the number of possible inputs for a printer is already larger. There is USB (and all its variants), older ports, wi-fi, bluetooth, infrared, memory cards. All input types are not present in every printer, different combinations are possible.

With the development of multifunctions printers, these tools are becoming more complex. To provide a fax function, the printer must be able to give and receive phone calls, which requires a phone line input. When building the software, it is necessary to consider the links between those functions. The complexity of requirements on the software may become high. As a result, it is necessary to have a mechanism that can handle these combinations: managing the variability. In this example, the complexity is low, but in real cases, the complexity may increase. Therefore, software product line engineering provides mechanisms to model the dependencies and constraints between the functions

2.1.2.SPL Challenges

Software product lines are designed to produce several similar software products. Their purpose is to reduce the time, money and complexity of their development. One method used is to take advantage of the commonality and re-use code pieces[2]. The second one is to manage the variability, to understand how the software products differ, and deal with their complexity.

Computer scientists have since 1960 understood the necessity of reducing the amount of code written and have provided different mechanisms. These mechanisms have been summarized by Clements *et al.* [5], and will be briefly explained in the next paragraphs.

---

[1] Commonality only refers to features always present. Here it is used as an approximation to better understand the software product line.

[2] The term *code piece* is used as a level independent term for modules, objects, methods...

The first practice used is called subroutines. Dahl *et al.* in the book '*Structured Programming*' [11] defines them by *«subroutine represents a primitive action».* The subroutine enabled developers to avoid duplicating several times the same action in the code. This mechanism was at a low level.

Then followed the modules. They achieved a task instead of an action, they were aimed at a higher level. Developers kept on developing different mechanisms at higher levels, with the object oriented methods. They encapsulated an object and its methods. They also proposed techniques to reduce the amount of code needed when an object had to be rewritten or slightly adapted. Finally, the developer used component based software engineering that was aimed at separating concerns in the software by being an *'interface-based programming'* [45].

These mechanisms are, however, more ad-hoc. They enable re-use only when there is an opportunity. They were not built for systematic re-use. After the code is implemented, a developer may re-use the object or the libraries somewhere else. The libraries are built for a specific use and may need to be adapted if the developer wants to re-use them. Software product line engineering is aimed at providing systematic re-use. The development of software products is done while keeping in mind how the code will be re-used later.

On top of that, the different mechanisms used to reduce amount of code, are mainly focused on the design and the implementation. In the software cycle development, implementation is only one process amongst others. There is the documentation of the software, the requirements, the tests units, etc. Like the source code, these documents will be similar. And by taking the benefit of commonality and the management of the variability, these processes will gain the same advantage of systematic re-usability.

2.1.3.SPL Engineering

In improving the re-use of software, software product line engineering emerged as a paradigm [29]. It addresses the issue of re-use in a specific context: the development of a set of software products that answer to similar requirements. The similarity is used in two ways: the commonality and the variability. Software Product Line engineering goes beyond the re-use of software elements, it also propose re-use in the documents and non-software artifacts. This section describes Software Product Line engineering and is inspired by a presentation made by P. Heymans *et al.* [14].

Software product line is a new paradigm [32], it provides new methods and techniques in the development of multiple related software systems. It builds on the re-use methods seen earlier, but goes beyond the design and implementation. Instead of being an opportunistic re-use, it becomes a programmed and systematic re-use.

Software product line addresses two kinds of software: individualized or mass product customized software. The first is aimed at making one product for each specific use. Considering a set of requirements given by a customer or a technician, the software is specifically adapted for them. This can be, for example, a car. When choosing options, a customer builds his own hardware and the software is adapted to that specific hardware. Mass product customized software is aimed for products that are different, but produced in large quantities. The mobile phone software is different from one model to another, but each occurrence of the model has the same software. The individualized software could be associated to the specific mission of one customer where the mass product is customized to the market segment through the market specific product.

Development

Now that the goal of software product line engineering has been established: developing and maintaining a set of similar software products [24] and enabling economies of scale in the production [29], it is necessary to learn how it is achieved. The next paragraphs will describe the differences between the regular software engineering and software product line engineering.

The products are similar and therefore, similarities are used to reduce the amount of work while differences must be managed. The similarity of the products engineered is called the commonality and the differences is named the variability. Coplien *et al.* [6] defines the commonality and the variability in term of assumptions. The two differ in

the fact that the commonality is true for each product and the variability is true in some of the products.

Software product line engineering aims at a systematic re-use. Therefore, the development is remodelled with two new activities. During the first, the re-use is planned and implemented. In the second activity, the desired product is built. These are, respectively the domain engineering and the application engineering. The next sections will detail these two steps. In each of these activities, the developers use artifacts. The artifacts are all the elements necessary in the development and include software pieces, requirements, models and tests units.

An artifact contains variability mechanism and is associated with a set of features. When a feature is selected for a product, the associated artifact is selected and mechanisms are used to activate the features in the artifacts.

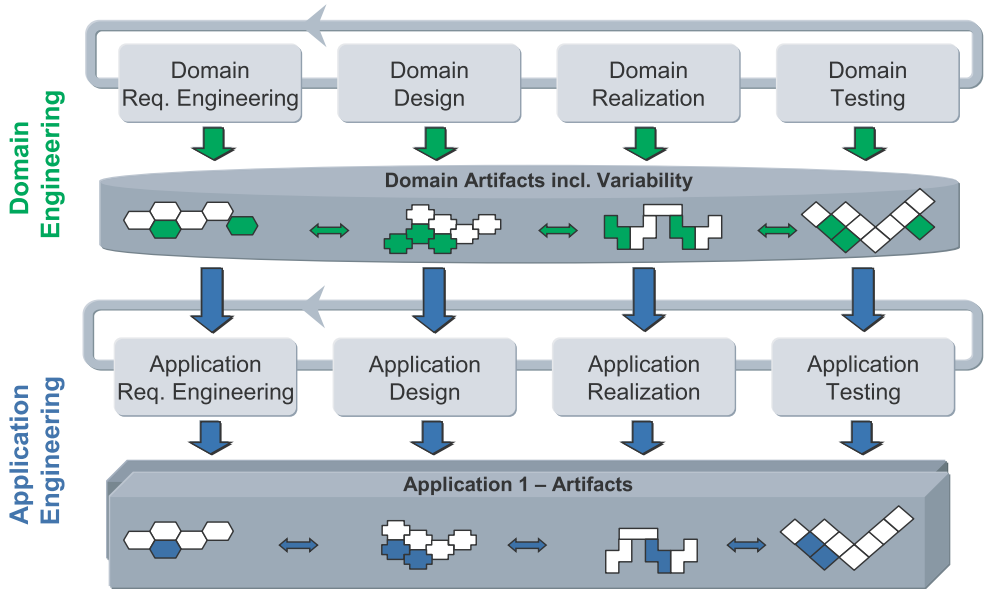


Figure 2.1: Software product line processes [32]

During the domain engineering activity, the artifacts are built for the whole set of products. Exploiting the commonality, the number of artifacts needed for the whole software product line is reduced and the time needed too. This activity is called the development for re-use. The artifacts are built but not used immediately allowing a larger number of possible products due to the postponement of product decisions.. These decision may be design decisions, requirements decisions… Each decision is built into the artifacts as a variation point [44]. This is the development of the global software.

The application engineering activity focuses on building one specific product. By using the artifacts of the domain engineering activity, the developer instantiate choices and generates the product. This is development with re-use, the customization of the global software.

Both domain engineering and application engineering are split into the same processes: requirements engineering, design, realization and testing. These steps are related in each activity. The artifacts built in a process of the domain activities are used in the corresponding process in the application engineering. The processes are similar to those in the regular software development cycle. They add the management of variability in the artifact produced.

2.1.4. Advantage of Software Product Line

Software product line engineering has been adopted by several companies because it provides them with advantages compared to regular software engineering. There are two main reasons behind the advantage of the software product line: the re-use, and the business model.

- Re-use

Software product line addresses development for a set of products. It improves re-use by aiming at a systematic re-use of source code and the documents accompanying it. By reusing and taking advantage of the commonality, the amount of work needed for development is reduced. This helps reduce the time needed and the cost of producing the software products. C.W. Krueger [23] listed different sources which discuss advantages that the software product line can bring in term of time to market, cost and meeting deadlines.

- Business model

As C. W. Krueger says in *Software Mass Customization* [24] software product lines is also a business model. The choice of customizing software is a business decision as is the scope of the software product line. Offering customization to clients is a business decision and not a technical decision. Krueger also gives some examples «*Real world success stories of software mass customization come from diverse areas such as mobile phones, e-commerce software, RAID storage systems, computer printers, diesel engines, telecom networks, enterprise software, construction and mining equipment, cars, ships, and airplanes.*» [24].

2.2. Variability and Commonality

Until now, the software product line has been described as taking advantage of re-use through commonality and has also been associated to the management of variability. This section will describe how the commonality and variability are modelled.

Software product line engineering enables the development of several products through delaying each decision. Van Gurp *et al.* [44] says: «*Each decision constrains the number of possible systems».* By delaying these decisions, the number of possible products is increased. A decision which is not made is kept as a *variation point*. In the same article, Van Gurp *et al.* [44] lists the levels where a variation point can be present:

- Requirement specification
- Architecture description
- Design documentation
- Source code
- Compiled code
- Linked code
- Running code

For each of these levels, an adapted mechanism provides the management of variability. As this work only focuses on the source code, only these mechanisms will be explained.

2.2.1. Feature Modeling

A variation point is a decision that is not made. Presenting each decision to the decider of the product may not be suitable as they may be numerous or in a technical language that the customer may not understand. As a result, it is necessary to have an abstraction for all the choices. The term *feature* is used. Kang *et al.* [17] describes the feature as *«A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems».* The feature is an abstraction for the developer and for the user.

The domain can be seen as all the features possible where a product is an arrangement of features. A product is built by selecting features wanted and unselecting the unwanted. A feature may be scattered among several artifacts and in the artifacts. A feature simplifies the choice for the user by reducing the number of choices he needs to make and the work of the developer by providing an abstraction that binds several portions of artifacts. A feature may be present at different levels as seen in Van Gurp levels, and several times in the same level.

Features can be categorized into several kinds. A typology has been made by Van Gurp *et al.* [44] and is reproduced here.

- Mandatory: A mandatory feature is a feature that is present in every product through the fact that it *«identifies a product»* [44].
- Optional: An optional feature is a feature that is not always present. It *«adds some value»* [44].
- Variant: A variant is an abstraction for a set of features.

Features are not always independent. A feature may require another one. For example, if you want a fax in your printer you also need a phone line input. Features have requirements, dependencies or constraints that need to be documented as well. Knowing these constraints and formalizing them enables tools to reason on them and reduce the work of the user while selecting features.

The last paragraph talked about the problem of documenting the features and their relations. A feature model(FM) describes the feature and their relations. A feature diagram is the graphic representation of the feature modelling. The feature diagram needs a semantic to enable both the users and the developers to agree on what signifies a diagram. This section describes the semantic of feature diagrams that is used in this work. To describe the feature diagram it uses a meta-model developed by Schobbens *et al.* [35]. A feature diagram is a model of the software product line. It describes the features, the relations between the features and the constraints on the selection of features.

Following is a feature diagram using this syntax. A feature diagram is a direct acyclic graph. The root is the root feature(*Software*). It's a mandatory feature. The nodes that are linked to a feature are its children (*radio, television,* etc are the children of *Software*). A child feature can't be selected if the father isn't selected. A feature that is optional is marked with a circle on its rectangle (*dualline* and *singleline* are both optional features).

Associated to a father, the relation between its children determines the constraint of their selection. There are three types of relations: *and, or* and *xor* (some models proposes to use cardinality instead, *or* becomes [1-n], *and* [n-n], *xor* [1-1] or any other value). *Software* has an *or* relation. This means that at least one of its children must be selected. The relation can be translated into a constraint: *radio* or *television* or *internet* or *telephone*. For a configuration to be valid, each constraint must be true when a feature is set to true in the constraint if selected. *Singleline* defines an *xor* relation. Exactly one child must be selected. Its constraint is: *adsl1* xor *adsl2* xor *vdsl.* The *and* relation is represented by default without any graphical adjunction, this is the case for *internet*: *dualline* and *singleline*.

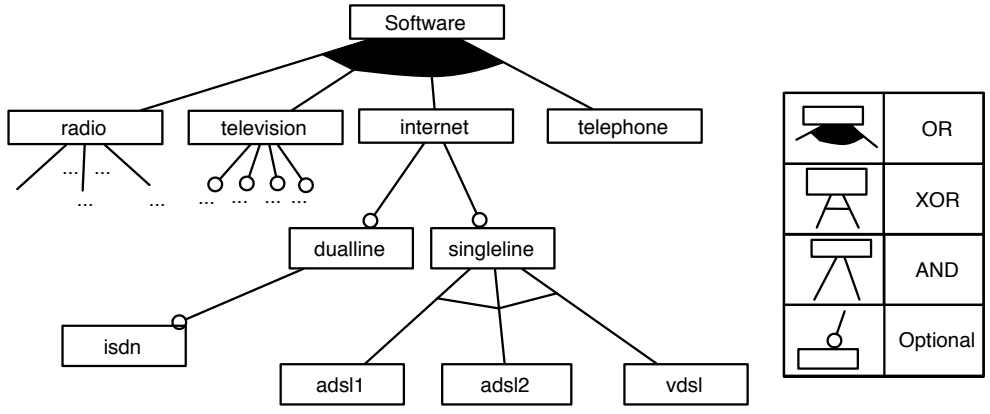


Figure 2.2: Feature Diagram

In this example (cf. *figure 2.2*), the root is *Software*. At least one of the features *radio, television, internet* and *telephone* must be selected. Every children of the root can be selected. If *radio* is selected, then each of its children is selected too because they are in an *and* relation. In the case of *singleline* only one child feature can be selected.

Additional constraints may be present through textual constraints (some models may impose conjunctive normal form to ease computation) and are not represented in the tree. For example, it can serve to express the fact that when a feature is selected another

must be selected too. Some of these constraints can also be represented graphically instead. Some models propose to use *requires* and *excludes* relations. A feature *A requires* feature *B*, if *B* must be selected when *A* is selected. A feature *A excludes* a feature *B*, if *A* can't be selected when *B* is selected.

However, more advanced feature diagrams have been studied by Schobbens *et al.* [35]. Instead of using logical relation, cardinality can be used to express the number of children that must be selected. For example, if a feature has 3 children, then the cardinality of an *and* would be [3..3]. Benavides *et al.* [2] introduced the notion of the *attribute* of a feature.They can add values to nodes and be used for calculations (for example: the cost of a solution associated to a product).

Each product that can be generated is based on a set of selected features (a set of functionalities). Selecting and unselecting features is the configuration process. Selecting a feature means, it will be present in the generated product, while unselecting a feature ensures the feature will not be present. Depending on the constraints of the feature model, selections cannot be arbitrary. In a valid configuration each feature is either selected or unselected and they respect the feature model constraints.

2.3. SPL Implementation

Software product line engineering aims at producing several products. Using artifacts, the specified products are generated. This is the realization process, which is not limited to the implementation. In the next section the two approaches will be described and compared. These approaches were highlighted by Kästner *et al.* [18], they are the compositional and the annotative approaches. This section now focuses on the realization process as it is in the scope of this work. The hole section uses the ideas and comparison done by Kästner *et al.* [18].

Kästner *et al.* [18] have compared approaches by using a list of criteria. However, they didn't do it as a way to prove that any approach is better than another but as way to use both approaches in a project.

- Feature traceability: The feature traceability is the ability for a developer to search where each feature is present in the artifacts.
- Modularity: The modularity is the capacity to use a feature, as it is, in a different product. For example, by being able to compile separately the feature, the developer can re-use the feature in another context.
- Granularity: The granularity of the software product line has been widely discussed in a paper by Kästner *et al.* [20]. The granularity refers to coarse-grained features implementation.
- Safety: The correctness of the every products in terms of syntax and type[3].
- Languages independence: The approach is independent from the language.
- Software product line adoption: Facility to adopt software product lines with existing code.

2.3.1. Compositional Approaches

Kästner *et al.* [18] describe the compositional approaches as «*Compositional approaches implement features as distinct (physically separated) code units. To generate a product line member for a feature selection, the corresponding code units are determined and composed, usually at compile-time or deploy-time.* ». In their article, they also give some examples of techniques using this approach. Building one product is done by selecting code units needed and then by linking them in that product.

[3] Concerning the type safety, the authors explain that some approaches independent of the implementation exists, however this work will also present an approach to help enforce the type safety in the case of our proposed approach

Advantages

- Traceability: Each feature is implemented as code unit, which provides an excellent traceability.
- Safety: This approach provides syntactically safe products, as each code unit must be syntactically safe and their composition does not modifies this property.

Disadvantages

- Modularity: Some approaches offer a separated compilation of each feature to modularize features.
- Granularity: The granularity is limited to the source code unit defined by the approach. A feature cannot be present at a lower level.
- Language independent: Providing a language independent compositional approach, requires some manual effort.
- Adoption: The adoption of software product lines has a great influence on existing code and development processes. This makes the adoption harder.

2.3.2. Annotative Approaches

Kästner et al. describe this approach as «*annotative approaches implement features with some form of explicit or implicit annotations in the source code. »* [18]. The features are scattered in the code. By associating features with portion of codes, it is then possible to build a product by keeping only the associated code active.

Developers already annotated software before software product lines, i.e. to help them understand and remember how the program works. With software product lines, annotations can, of course, be used to explore and understand code [21] and how features are implemented, but also to generate products. They annotate the artifacts with features to enable generation of products. How the artifact is transformed into different products is realized by using a generative approach to create products from this artifact.

For Czarnecki *et al.* [8] the key element of generative programming is the automation of manufacturing software systems given requirements, using reusable components. The generative approach is based on transformations applied to the artifact. Czarnecki *et al.* [10] describe a transformation as *«an automated, semantically correct (as opposed to arbitrary) modifications of a program. It is usually specified as an application of a* transformation *to a program at a certain location. In other words, a transform describes a generic modification of a program and a transformation is a specific instance of a transform.».* For example, a transformation can be used to perform dead-code elimination [10]. In the case of an annotative approach, the transformation can be used to generate different products. Generative programming rests on a configuration, a set of selected features, to manufacture a specific product. The automation can combine several features, i.e. different portions of source code, or make inactive the unselected features. Pruning approach is a generative approach. It removes the code instead of deactivating it [20]. The features are associated with portions of source code. Then by removing the code of undesired features, a new product is generated.

Advantages

- Granularity: The annotative approach enables a fine-grained implementation, as it depends on the underlying annotation structure, it may be up to the character or the line that is associated to a feature.
- Language independent: This approach is language independent as the underlying structure may be language independent or requires a little modification.
- Adoption: Does Not have an impact on the language and affects in some limits the development process which provides an easier adoption than compositional.

Disadvantages

- Traceability: Features are scattered through code, which makes traceability difficult. However, it is possible through the use of support tools.

- Modularity: This approach does not provide separated compilation.
- Safety: As the granularity of this approach can be as low as a character, it may produce incorrect products. However, depending on the annotation mechanism, the safety can be enforced.

2.3.3.Summary of approaches comparison

After learning what is SPL and main notions, this chapter studies the different approaches to implementing the SPL. The two approaches compared are summarized in the *table 1.*

	Compositional	Annotative
Traceability	+	-
Modularity	-	-
Granularity	-	+
Safety	+	-
Language independent	-	+
Adoption	-	+

Table 1: comparing implementation approaches

The *table 1* displays the disadvantages (-) versus the advantages (+). To fulfill the goal of a lightweight approach, an approach that does not impose changes in the design of the software, nor requires a language switch and could be easily adopted by developers. The annotative approach fulfils these goals. Two of its three disadvantages, the traceability and the safety can be improved by a tool support. The compositional approach imposes a large change in the design of the tool and the use of another language. They cannot be easily adopted by developers. These are the reasons why the annotative approach is adopted as a light way approach.

3. Annotative approaches

In the previous chapter, a quick survey of the software product line explained them. It also explained the two approaches available to develop software product lines. The annotative approach was taken based on the scope of this work, implementing software product lines in a light weight approach. It allows a fine-grained feature implementation, avoids separating feature into physically separated files, is language independent and therefore provides a lighter approach that can be more easily adopted. This chapter will focus on describing existing annotative approaches and different annotation mechanisms available.

The annotative approach, as its name says, is an approach where the developer uses annotation to implement features, the source code is annotated with features. The form of the *comment* can vary and it can be explicit or implicit. The annotation contains the information about which feature, the associated portion of source code is implementing.

First the chapter will discuss on code annotation and its different uses. In the second half, the use of tags to implement SPL will be discussed.

3.1. Code Annotation

Annotations comment the source code. They have been used for different purposes by developers. According to Cachopo [4], they allows separation in the case of crosscutting concerns. For example, they are used in the context of services architectures [31].

Pawlak in an article [31] describes how annotations can be useful for services architecture in meta-programming. With SPOON, a «Compile-time Annotation Processing for Middleware» [31], he proposes to use annotations to *raise a program's abstraction level*. By expressing intentions of a program, the developers can ensure the program follows them while the implementation may depend on constraints, like target environment. The user is working with abstract and declarative intentions. On top of raising the abstraction level, using annotations avoids redundant information. The deployment information is contained in the program instead of having a separate deployment information that would require repeatition of objects and components signatures to link them. Avoiding redundancy reduces the risk of errors when modifying the objects or components. In this context, annotations can be used to optimize the program. Depending on the target environment, some parts of the program can be made active, removed or modified by parameters.

Annotations explicit the intents of the source code. In the case of software product lines, this fact can be re-used to associate the features to the source code. The developers have already used different annotative approaches to enable re-use and adaptation of programs to different needs or target environments. These approaches are described in the next sections.

3.1.1. IFDEF

C language is present on almost every systems. It is used to develop multi-systems software. However, programs uses libraries and systems calls that may not be identical on every system. To ensure the software works on different systems, developers use #IFDEF as a portability mechanism [36]. An #IFDEF is an instrument that can be used to include some portion of source code only when given parameters are present. The pre-compiler uses variables defined by the target environment to activate or not the source code contained between the two markers (#IFDEF and #ENDIF). If the condition is false, then the #ELSE portion is made active. See *figure 3.1* for an example of code using #IFDEF.

The #IFDEF could be used to implement software product lines. Instead of giving information about the target environment, the IFEDEF could be used to delimitate the

features. Using a set of selected features, a pre-compiler could generate an artifact containing only the necessary code. However the #IFDEF mechanisms have some flaws that makes this solution not ideal for implementation.

As the article is studying the issue in the context of portability, Spencer *et al.* [36] states the issue of using this approach in the context of the portability. As long as there are only a small number of target environments, #IFDEF is suitable. With the increase in the number of systems on which the software is ported, the #IFDEF are becoming increasingly numerous. The code becomes less and less readable and difficultly maintainable. The #ELSE adds to the confusion. It becomes difficult to know how the pre-compiler is going to interpret the code if markers are used inside already marked portions with different features. In some cases, the authors have found several layers of markers embedded. Is this portion associated to this feature or will it be inactivated if an other feature is not selected? On top of this complexity, the developer using an opening and an ending markers, may miss-mark the source code, which would provoke erroneous pre-compiled code. The last problem of #IFDEF mechanism is that the markers are part of the logic of the program. They are mixed with instructions, making it even more difficult to understand the code.

```
/* name of this site */
#ifdef GETHOSTNAME
# # #else # #
# # # #
char *hostname; undef SITENAME define SITENAME hostname /* !GETHOSTNAME */
ifdef
DOUNAME include <sys/utsname.h> struct utsname utsn; undef SITENAME define
SITENAME utsn.nodename
else /* !DOUNAME */ ifdef PHOSTNAME
char *hostname; undef SITENAME define SITENAME hostname
undef SITENAME
# # # # # # # # # #endif /* GETHOSTNAME */
endif
endif /* PHOSTNAME */ /* DOUNAME */
else /* !PHOSTNAME */ ifdef WHOAMI
define SITENAME sysname endif /* WHOAMI *
```

Figure 3.1: IFDEF example [36]

These different problems led Krueger [24] to state that «*they are not manageable beyond a small number of product variations. Moreover they are code-level mechanisms that are ill-suited to express product-level constraints*». The #IFDEF mechanism is not an apropriate solution for the implementation of software product lines as it provides a complex solution. Even reflexions given by Spencer [36] trying to provide an answer to clarify #IFDEF are unsuitable in this case. He recommends limiting the use of #IFDEF to the declarations. This idea would greatly reduce the possibility of granularity. The developer would then need to design his program methods and objects according to features. This would strongly affect the development and would be against the light weight approach goal.

3.1.2.Frames

Frames are an older technology (1970s) using annotation. Loughran *et al.* [26] describe this as an approach transforming portions of source code into modules. Such approaches are named XVCL by Wong *et al.* and FPL by Sauer *et al*. To achieve this task, the source code is annotated and then a pre-processor realizes the transformation. It also uses tags to add metadata information to, for example, pinpoint location of variability mechanisms. The information can then be used to locate source code that needs to be modified in a different context. The goal is to isolate concerns, methods and classes in separate and hierarchical layers.

This approach is aimed at enabling re-use by identifying and separating variability locations. This approach wasn't conceived for a systematic re-use as it allows ad-hoc reusing of some portions of source code because they happened to be reusable. *«The lower order frames are the most reusable as they contain less context sensitive information»* [26]. The examples listed as reusable are not of great interest for this work as this approach is limited to re-use of *«IO routines, library functions etc...».* On top of this issue, frame tech-

nology mixes the frame annotation to the annotation of how frames relates to each others. This breaks the idea of separation of concerns, and the software product line engineering processes. See *figure 3.2* for an example of frame technology.

```
<x-frame name="x_CreateTask" language="java"> <set var="PACKAGE" value="BusinessLogic"/>
<break name="CREATETASK_NEW_PARAMETERS"/>
package <value-of expr="?@PACKAGE?"/> ...
import java.util.*;
<break name="CREATETASK_NEW_IMPORTS"/>
public class CADCreateTask { private Caller aCaller; private Task aTask;
<break name="CREATETASK_NEW_ATTRIBUTES"/>
public Caller GetCallerInfo() { . . .// code about capturing Caller's info
return aCaller;
} public int SaveTask() {
... . . .  // code about saving a task
<break name ="Validation"/> int nTaskID = aTask.Save();
return nTaskID;
}
<select option="CT-DISP"> <option value="SEPARATED">
<adapt x-frame = "InformDispatcher"/> </option>
</select>
<break name="CREATETASK_NEW_METHODS"/> } </x-frame>
```

Figure 3.2: Frame technology example [46]

3.1.3. Colored IDE (CIDE)

An other approach was created by Kästner *et al.* [21] to implement software product lines. Their approach relies on a coloured IDE (CIDE). The developer uses the IDE to annotate the program. The developer selects portion of source code and can annotate them with features. This information is then displayed by the means of colours. The use of colour for visualizing features will be described in the next chapter. There is no information added in the source code. Therefore the information is registered inside the IDE by modifying the abstract syntax tree of the source. The generation of the variants is enabled by exporting the annotated portion of source code into modules which are implemented by using a compositional approach. Their approach can be used to fill the gap between annotative and compositional approaches by using annotation to isolate features into modules. They also propose using the pruning of the source code to generate products. The pruning is done by removing every portion of source code that is not necessary in the selected product. A portion is removed if it there are no features associated that are selected.

They have studied two approaches, to save the annotation. The first was to save them in the code and remove them when the editor opens the file and displays it. They rejected this idea as it is an invasive technique and would allow the user to make manual changes to the annotation without respecting the rules. The second one is to use a separate file to save them. It is not invasive in the code but forces the developer to use an adapted IDE if he does not want to loose the annotation. As the code is not changed, it provides a safe approach with legacy code which is not modified.

As Kästner *et al.* [21] say, the annotation are not only used to explore and understand the source code and the implementation of features. They are also used to generate products, or variants as they are called in the paper. The author believes that if the annotation activity is also used to generate products, it implicates the developer in this activity more than when it is only used for navigation. Even if documenting software is a best practice, some developers are not doing it thoroughly. The completeness of the annotation is then *« simply ensured by the fact that generated variants are compiled and tested in the normal development process»* [21].

While the annotative approach was described in the chapter on software product lines, one advantage of this approach is the granularity. The developer can choose to annotate only some lines or some characters and is not limited to physically separated modules. While this advantage reduces the need to rethink the program, it may be source of errors. As it was pointed out in the #IFDEF approach, allowing the developer place the end of the annotation in any place may itself result in incorrect annotation. Therefore the IDE enforces «*disciplined annotations*» [21]. CIDE uses the «*underlying code struc-*

ture» to forbid «*arbitrary annotation»: «only structural elements of the code, e.g., classes, methods, statements or even parameters».* The underlying code structure is the abstract syntax tree. Only nodes of the abstract syntax tree(AST) can be annotated. However, as for the annotation itself, the enforcement of correct annotation relies on the tool itself. To ensure the safety of the approach, CIDE not only limits annotation to structural elements, the nodes of the AST, it also restrains annotation to mandatory nodes of the abstract syntax tree. Kästner *et al.* describe mandatory nodes as opposed to optional nodes that *«can be removed without invalidating the syntax»* [21].

This approach highlights the necessity of a non arbitrary annotation to ensure safety. It also separates the annotation from the logic of the code to provide a clear code. Therefore, it proposes a realistic approach to implement a software products line. However, the necessity of a specific tool to enforce rules and to save the annotation is opposed to the light weight approach desired, as changing from an IDE to another may not be possible in every use case.

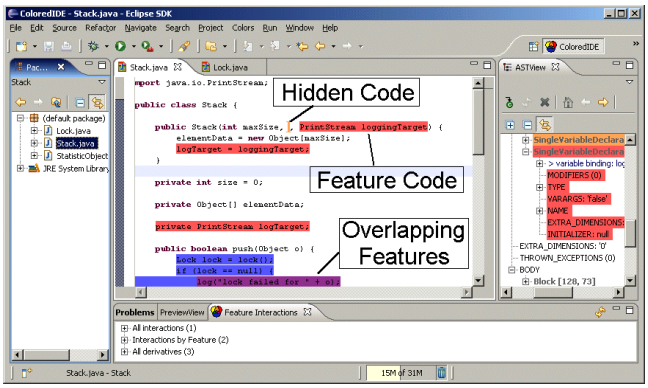


Figure 3.3: CIDE [20]

3.2. Tags

Tags were present and used before Web2.0. However with the development of websites like Flickr, an other form of annotation has been popularized: tags. Users uploading photos can tag pictures to help others people search and to help remind themselves of information about the photos. By using this context, Ryall [34] explained his research on source code annotation. This section will describe tags, their use in both the social and technical domains, how and why they are used. Finally the section will end with a short description of how they can help to implement software product lines.

Tag is a keyword associated by the user. The user choose words that he finds relevant to the annotated object. Rubbani has studied tags in the context of semantic web and news and says that tags can be used as a classification and *«Tags could be anything, like personality names involved in news, news nature like funny, excited, sad (death, accidents, terrorism) etc or actions like win, loose, etc»* [33]. In the context of the news, tags even emerge as a new data type.

There are different forms of tags, depending on the context of their use. The tags are used in several domains. The different domains were studied by Ryall [34]. They are the social tagging and software development. Two kind of annotations can be distinguished: structured and unstructured comments.

Ryall identified several goals that annotation could help to achieve [34], first one is reminding, second one is re-finding. Annotation, in all the contexts studied, can help the user to remember information about the annotated object shortly after writing it or even for longer periods afterwards.. In the context of development, documentation helps understand what it the purpose of a portion of source code, or why it has been conceived this way. The second purpose is re-finding. Adding information on objects, enables the user to more easily find an object that may not be explicit. It may be hard to find a method responsible for a specific task when it only deals with variables and instructions without explaining its role.

3.2.1. Where are Tags Used?

Treude *et al.* [43] gives a reason for the success of tags, *«The success of tags is closely related to their bottom-up nature: tags do not have to be pre-defined, every user can choose their own tags, and the number of tags per item is arbitrary.»* They also give interest in tags *«Mainly as a kind of categorization. [...] Tags are useful for identifying crosscutting concerns like performance or accessibility or scalability or responsiveness, things like that, or testing.»* The fact that users are free to use their own tags contributes to the success of tagging. There is no need to remember a specific vocabulary and nor to be consistent between tagging.

Ryall has studied social tagging [34] and according to him, users tag their photos for two reasons. Tagging helps them find their pictures later. They can also have a more social goal; as a way of communicating with others. The author also noted that users are free to use their own keywords to associate with the objects, but the vocabulary tends to converge on a common terminology. Tagging forms a bottom-up approach. As a result, the tags define *«semi-structured information spaces that are often referred to as 'social classifications'»* [40] Tags are unstructured annotations.

Tags are also used to generate information about the source code. However, this approach is a particular case as tags are automatically generated and managed. Nonetheless, it indicates the interest of tagging for automation. Horwitz proposes a method to analyze the differences between two versions of a program[15]. Each component of the software receives a unique tag. Tags are automatically managed, a new one being created when the developer adds a component, removed when the component disappears and it keeps the link between the tag and the component when it is moved or modified. By using these tags, Horwitz was able to compare each component between two versions and determine through a method, if it is a semantical change.

The advantage of this method, is that it links the tag to the element of the source code and uses the source code to make computations. Tags no longer use a user-defined vocabulary, but are automatically generated. The tags are only a way to add meta information to the source code to help with its automation.

The developers using Java may be aware that an annotation system can be used to help them comment their code[30]. Using structured comments and tags like *@author* and *@version* enables automation of the creation of the documentation [25]. The developer, as a best-practice, documents his code while writing it. He uses these tags and keywords to add metadata. For example «*@author CGauthier*» could indicate that the author of the class is named C. Gauthier. Then a tool is used to parse the Java files and their folders. The information extracted is linked to the structure of the Java code to produce a useful documentation. If the code is changed, as long as the comments are still correct, it can automatically be used to create a new correct and updated documentation. Developers can even use their own tags with a doclet [22], a *«set of Java classes that generates customized documentation from the Javadoc tool»*

However, contrarily to the CIDE approach, the developer does not indicate a specific range of associations with the source code for the JavaDoc. As classical comments, that are unstructured, *«the proximity between these two elements usually forms an implicit link»* [34]. It is enough to write the comment next to an element of the Java structure to have them associated. Therefore there is no need to check if the annotation is correctly englobing a node of the abstract syntax tree as the next node is automatically annotated. Ryall by quoting Kaebling [16] highlights the fact that this link is implicit instead of being explicit with the help of what he calls *scoped comments*.

Besides documenting source code, the Java annotation mechanism can also be used to navigate through the program. By using tags to send the developer to an other part of the code and by linking portions of source code with, respectively, *@see* and *@link*, the developer enables a path to navigate in the source code. The navigation helps the developer to understand how the program is built. However, this activity is a supplementary activity. As it not intended a short term goal for the developer who is writing them, he may not be motivated to write them.

Ryall also identified the possibility of using Java annotations to modify how the source code is compiled and runt. For example, by tagging files, the developer can indicate how the file must be handled by the source code management system [34]. In this context, the developer can have a better opportunity to control how his program works.

3.2.2. TagSEA, Tags for Software Engineering

Ryall used a tool to study how developers used annotations to facilitate development. This tool was developed by the CHISEL team with Storey *et al.* The tool is called TagSEA. TagSEA is an Eclipse plugin to manage tags. Developers can add tags in the source code. TagSEA parses files to manage the tags and provides the developers with tools to find tags and their locations in the program (see *figure 3.4* for a screenshot of TagSEA).

Ryall mentioned two goals of TagSEA: *«providing more structure for organizing annotations by allowing developers to define their own vocabulary, and allowing developers to link crosscutting concerns in the software»* [34]. TagSEA enables the use of any keywords by the developer as opposed to the limited support of specific standard words in standard by the IDE. TagSEA also enables use of hierarchical tags as a way to organize the keywords.

TagSEA, by providing tools to locate tags efficiently, enables the developer to link different locations in the source code under the same keyword. It therefore helps the developer in linking concerns that are scattered in the source code: crosscutting concerns. TagSEA makes the annotation of source code with tags easier by providing tools to tag, re-find and manage tags. The next paragraphs describe TagSEA and are summarized from two publications by Storey *et al.* [40, 41].

Tags are annotations which are located in comments. They begin with a @tag and are followed with the keywords, metadata like the author and the date and a message. In this example, the tag is *bug*.

```
//@tag bug -author="CG" -date="enCA:18/01/07" : Crash when reset
```

This example of tagging is a comment (//). The tag used is named *bug*. The metadata are the author name (-author=) and the date (-date) which uses the time zone. Finally a message is added (:). Hierarchical tags have the form of *system.subsystem.crash.* TagSEA provides an interface to display the hierarchy of tags, their position. The location gives information about the line number, the file, the associated element of the source code associated, the author and the date. It also provides a visualization under the form of a tag clouds where tags are bigger if they are used more.

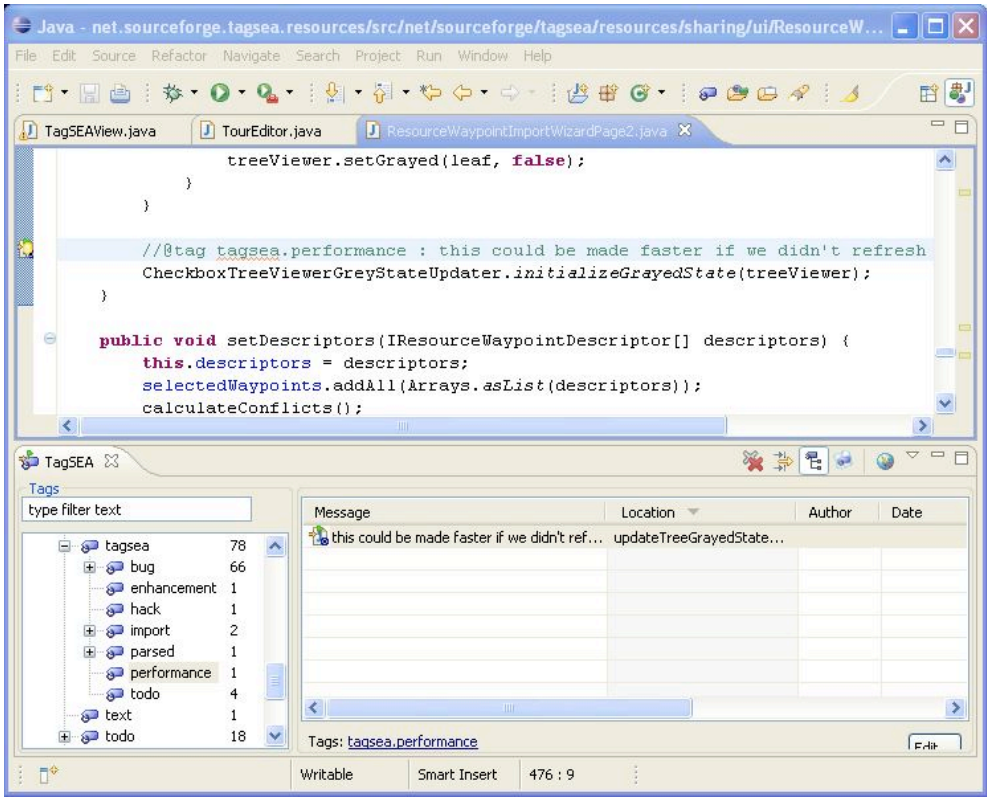


Figure 3.4: Screen capture of TagSEA from TagSEA Website

TagSEA uses the metaphor of waypoints. Each tag has a location which acts as a waypoint. Extracted from the navigation vocabulary, a waypoint is a point of interest. Like navigation systems, a set of waypoints can be transformed into a route. Routes are seen as a way of navigating through code and exploring it.

In the context of software product lines, the use of route to explore how a feature is implemented could help the user. It prevents him from the necessity of searching and going to each location manually. As Storey *et al.* [40] say: *« Routes are sequences of waypoints to specific code features [...]».*

As the tags are present in the source code, every developer can access them independently of any IDE. By using tags to communicate with others developers, TagSEA helps coordination in the development team. For example, a developer can easily mark a region he knows to be a potential source of bugs to be checked by others developers. Tags can also be used to capture knowledge. As developers implement, they can record information about the source code. It is quicker to use keywords to classify knowledge. They can then use a message to give more detail. This information can then be accessed more easily as it is not necessary to search using text search tools. The developers only need to use the lists of hierarchical tags to search locations and messages.

TagSEA enable other plugins to define their own tags. Therefore it does not limit the possibility of its use. Developers can define their own tag syntax and use it to expand the possibility of tagging. TagSEA also allows the definition of new supported languages in the same process.

3.2.3. Tags for Software Product Lines

As stated in the introduction, the goal of this work is to provide a light weight approach to implementing software product lines. Two approaches exist for implementation: the compositional and the annotative. As the compositional requires the adaptation of the software by dividing software into modules, it would have too much of an effect on the implementation by requiring a paradigm shift. The annotative approach provided a better solution as it is only necessary to add information about features to the code. Although this approach can also have an impact on the implementation, it is close to practices that are already present like documenting source code.

Different annotation systems and annotative approaches were studied. Some approaches were rejected as they were impractical if the software product line was non trivial. Two approaches provided more interest, the CIDE and the frame technology. Due to its conception, the frame approach mixed features and their use rules. However, its use of tags to add metadata about features was an innovative idea. The CIDE approach provided a systematic re-use approach as it had been conceived to implement software product lines and was a complete approach as it enabled generation of variants using the annotation. Nonetheless, it was based on a specific IDE, which would break the light weight approach goal.

As tags are already used, they provide a better tool support and have already been accepted by the developers. They can thus provide a light weight approach to implementing software product lines using an annotative approach. They also provide a separation between the code itself and the features which was lacking in *Frame technology.*

3.3. Adopted SPL Tagging Approach

Different authors proposed the annotative approaches but it does not fulfill the goal of this work. Frames mix annotations and models. CIDE is dependent on an adapted IDE, #IFDEF is the source of complex and unreadable source code and frames is not adapted to the systematic re-use. These annotation approaches are just hiding the composition between annotation. There is an important necessity for tools that are specifically adapted for these approaches, like IDE or preprocessor, as they do not just generate variants, but transform the program into software modules which then need to be composed.

The tagging approach uses tags to annotates source code. A tag represents a feature from the feature model. As existing approaches were judged insufficient, the tags appeared as an interesting opportunity. Boucher *et al.* [3] propose to use tags with a specific keyword followed by the name of features associated. A tag represents a feature

from the feature model. The specific tag can then be used by the developer as an indication of the fact that a portion of source code is annotated. By reading the names of the features, he could then known the features associated.

To provide an approach independent of any language, they propose to put tags in comments, as comments are present in almost every languages. This offers the advantage of separating the annotation from the logic of the program while still having it in the source code and not separated from the file. In the next example, the feature called *featureD* from the FD (see *figure 3.6*) is tagged (see *figure 3.5).*

```
/*@feature:root.featureA.featureD@*/
```

Figure 3.5: Tagging of featureD

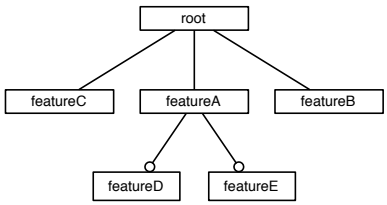


Figure 3.6: Feature diagram

```
/*@feature:root.featureA:root.featureC@*/
/*@feature:root.featureB@*//*@!file!@*/
```

Figure 3.7: Two kinds of tagging

The two comments of *figure 3.7* are examples of tagging used to annotate source code with features. It is inside a comment to be still displayed in the source code but apart from the logic of the program. The syntax used in this example is the one used in this work (however, it is only one of the numerous possible tag syntaxes possible). There are two markers that indicate that this comment is used for features: @feature: and @. The first is the begin marker and the other is the end marker of the tagging. The two features are represented by the name of the features, which will be described in the next chapter. They are separated by colons. To tag several features, their name can be put in the same tagging and separated with a colon. The first tagging means that the next element of the source code is associated with two features that have for name *featureA* and *featureC*, children of feature *root*. A special tags can also be used to indicate that the entire file is associated with the feature. In the second example, the feature called *featureNameB* is associated with the entire file.

The association between the tag and the portion of source code uses a similar idea as the one in Javadoc, the proximity [34]. The closest following element of the AST is associated to the features tagged. As one or more features can be used as tags, the portion of source code can be associated with more than one feature at a time.

```
public static void main(String[] args) {

      if(test=true) {
            /*@feature:root.featureA@*/
            doSomethingForCar();
            doSomethingElse();
      }
}
```

Figure 3.8 Tagging association

In this example (*figure 3.8)*, a feature called *featureA* is associated with the call of a method *doSomethingForCar* (dashed red box). The next element of the abstract tree is associated with the features. Although, it may look simple, it is necessary to know exactly with which portion of source code the tagging is going to be associated. The tagging approach uses the idea cited by Ryall [34] of *scoped comments.* As comments are

scoped using a semantic, the developer need to mentally compute the association. However, if the association was displayed, it would facilitate its work and reassure him.

The elements that are associated are nodes of the AST, as in the approach taken by Kästner *et al.* [21]. However, this approach does not use an ending marker, but only a tag marking the beginning of the annotated portion. The end of the annotation is automatically the end of the associated node. The approach does not allow arbitrary annotation as only nodes of the AST can be annotated.

The success of tags also originates from the freedom that the user has to choose for tags. However, in the tagging approach, the developer cannot choose freely which keywords he wants to use. He has to use names of the related features. These names come from the feature diagram used to represent the software product line.

If their approach cannot use this advantage, there is another reason why the annotation will be used by the developer. As Kästner *et al.* [21] stated, the annotation is used to generate products. According to them, this is a sufficient reason to ensure the annotation is done. On top of that, as the tagging approach does not require the use of any tool to annotate, the developer can do it while coding, without having to use the mouse to call for a function of annotation and select the code annotated; it reduces the weight of such an approach.

The JavaDoc analogy was used to explain the approach because tagging features is a closed process, the same way that the developer writes comments while writing code, he can write tags during the same period. Like the comments, they also explicit the intention of the code. The developer could also tag the source code after it has been written. For example with legacy code that must be transformed into a software product line, an existing software can be tagged.

The tagging approach is independent of any tool, as it saves the annotation in the code. Any IDE can open the annotated files without losing the annotations. This reduces the impact of the tagging approach compared to IDE-specific approaches. One of the reasons CIDE stored annotations independently is that, their approach used rules to enforce correct annotation, i.e. annotating only nodes of the AST. As tagging does not need to be rule enforced to annotate only nodes of the AST, there is no need for tools to annotate features with tags.

However, if one of the advantages of the tagging approach is to not require a tool, it allows developers to select nodes that Kästner *et al.* [21] qualified as *optional*. Therefore it could produce a syntactically erroneous product. The approach simply relies on the developer to avoid such errors, like tagging only the *then* clause of an *if* statement.

In the tagging approach, tagging is only one part. Tags serve to annotate source code. The annotation contains the features which are implemented by the annotated portion of source code. The second part, is the generation of a product. The annotative information can be used to generate a specific product according to given specifications. The specification is called a configuration, it is a set of features that are selected for a product. Only the source code that is associated with these features or that is not associated with any feature should be active in the generated product. The solution they took in this approach is a transformation of the code: the pruning.

To render inactive the source code only associated with non-selected features, a tool prunes this code. This part of the approach does require a tool. The tool parses files and searches for tags. It then associates the portion of source code with the features named in the tags. Given a set of selected features, the portions of program that are only associated with unselected features are pruned. In the *figure 3.9*, the original source code is pruned with a configuration where the feature *root.featureA* is not selected. Therefore the code associated (dashed red box) is removed in the pruned version on the right. Technically, the node of the AST is removed.

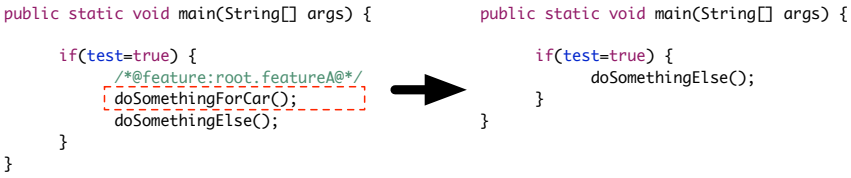


Figure 3.9: Code pruning

As the scope of each annotation is a node of the abstract tree, the pruning is realized directly on the node. Modifying the abstract syntax tree instead of the source code directly has some advantages as Czarnecki *et al.* [10] says. One of them is that the automatic refactoring or modifications are easier if the code is already as an abstract syntax tree.

The pruning is done after the code is resolved into an abstract syntax tree by a *pruner*. As nodes of the tree are removed, the abstract syntax tree is transformed by *rewrite rules* back into a new source code by an *unparser* [10]. Then the pruned code can follow his path through regular compilation. If the tags are annotating an entire file, then the file itself is deleted from the generated project. The resulting product is a compiled program containing only the code of selected features.

An advantage of not requiring any tool to annotate the feature is that, the annotated source code could be used for other purposes than pruning. Other methods like compositional methods could be used to transform the source code. As long as the tool uses the same semantic to associate tags with source code, the developer could tag once and use the result several times.

As already mentioned, the approach is tool independent at least for annotating source code with tags. However this requirement has a flaw with the syntactic correctness. As seen, this translates into allowing mandatory nodes to be annotated. If these nodes are pruned, then the resulting code is syntactically incorrect. These errors are easy to locate by the developers who can correct them quickly. The problem is that they may only appear when a certain set of features is not selected. This issue may be solved by using a tool as a support to the tagging.

Boucher *et al.* [3] chose the pruning for their approach as it was the technique that was best suited to the constraints given in the original issue. The company that developed the approach, expressed the desire to switch to software product lines but couldn't change the language they used. On top of that, software embedded in satellites, has strong requirements in term of space and memory allocated.

Therefore, pruning the unnecessary code lowers the size of the final program. Tagging variables with features reduces the memory to only what is strictly necessary, as variable allocation requests are pruned from the source code when they are not necessary.

Pruning can be applied to entire files. Therefore, a product could use a variable without it being defined in the pruned product and produce a type error. The source of the error may be easy to find, but as already noticed with the syntactic errors, they may not produce themselves before a specific configuration is pruned. The solution would be to generate every possible product and check if they are correctly typed and do not contain any syntactic errors. However the number of possible configurations may be too large to do this way.

Boucher *et al.* [3] proposed to use the *principal ancestors*, a set of features that will always be present with a given feature. By ensuring that *«Each feature can only use variables, functions and types declared by itself or by its principal dependencies. »* [3], the developer can ensure the safety of the type. The developer only needs to prune the software once for each feature by selecting only the features and its principal dependencies and checking their correctness.

To compute the *principal ancestors,* they propose a heuristic. Given an origin feature, the set will contain its mandatory siblings, its father and all features imposed by required constraints. For each feature in the set, it is necessary to repeat the process iteratively.

When the set is stabilized, only features that will always be present in legal configuration will be present[1].

The advantage of the heuristic is that it does not require complex tools to resolve the feature diagram. However, some features may be missed. It is also possible to compute an exact computation of the features that will always be present with the origin feature by resolving the feature model in the case where the feature is selected. This resolution uses both the constraints of the tree, the kind of relations between nodes and the constraints. As this algorithm requires to use a semantic approach of the feature model, it may take more time than the heuristic in the case of important feature models. For clarity purposes, it will be called in the rest of the thesis the minimal set of features (associated with a feature).

[1] A legal configuration is a configuration which respects rules and constraints of the feature model and where every feature is either selected or unselected

4. Features Visualization

Development is a complex cognitive task. To support it, best-practices like documentation were created. Documents produced can then help the developer to build a mental model, to understand the software [38]. In this process visualization can help achieve this task. Visualization are numerous and dependent on different criterions. It is not possible to present all the different visualizations possible in a chapter and this is not the goal of this work. Instead, a few visualizations that were adapted for this work.

4.1. MetaModel Visualization

Heidenreich *et al.* [13] described a visualization technique to help the developers in the context of annotative approaches. Its goal is to link features with their implementation. For that purpose he proposes a visualization technique based on four views and describes them in the article independently of any realization method.

- Realization View
- Variant View
- Context View
- Property-Changes View

The realization view links one features to the software elements that implement it. This view displays only the software elements that are annotated with the feature. The purpose is to help the developer to understand the complexity of a feature through the number of elements that are linked to it. It also helps to check the nature of coarse-grain elements that implement the feature. In *figure 4.1*, a model is used and its elements are greyed if they are not associated with the selected feature.

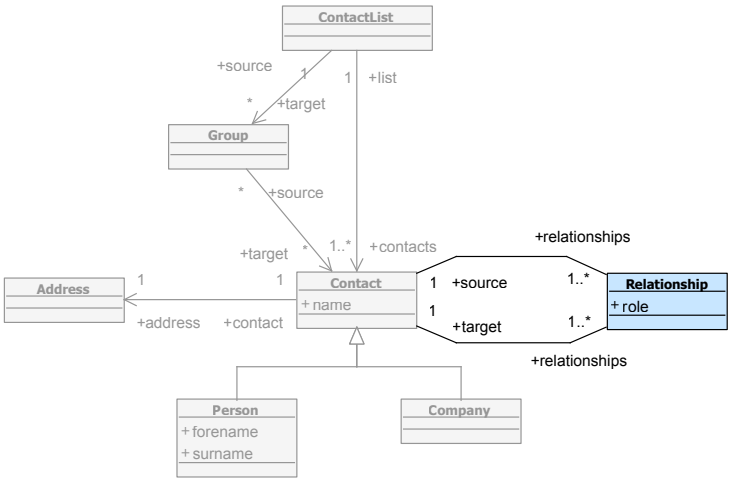


Figure 4.1: Realization view on a diagram

The variant view displays the elements as they would be in a variant, a product. Using a configuration, all the elements that won't appear in the related product are hidden. This view helps in analyzing if the resulting product is correct and in seeing the impacts made by changes. Heidenreich *et al.* do not indicate if the elements that are not linked to any feature should be displayed. The *figure 4.2* shows an example of a variant view.

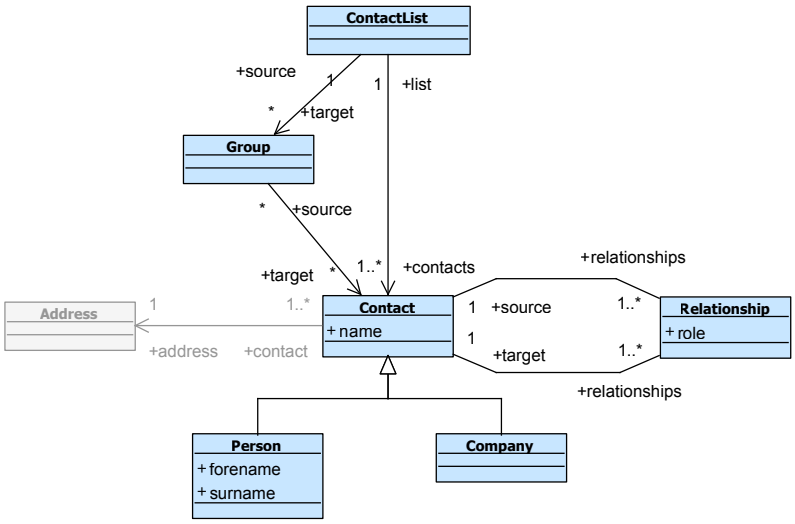


Figure 4.2: Variant view on a diagram

The context view reposes on a similar idea as the realization view except it addresses the issue in the case of more than one selected features. Each feature selected is associated with a color (see *Figure 4.3* for an example of color associated with features). Then the annotated source code is colored according to its feature. For the authors, this view enable the perception of the region where features are interlaced and of relations between features implementation.

Figure 4.3: Colors associated to the features [13]

The *figure 4.3* is a feature diagram. Three features each have a different colour associated. The same colour will be used to highlight features in source code.

The last view is aimed at solving the issue of changes made to the structure and properties of an artifact. The authors propose to use an eye-catching colour to indicate the location of these spots. The goal of this view is to help the developer to identify properties that may change depending on the features selected.

Heidenreich *et al.* [13] identified different questions their tool could help to answer, they are reproduced here from their work:

- *«Which elements implement a specific feature? »* is answered by the Realization view.
- *«Which elements are used in a specified product? »* is answered by the Variant view.
- *«Is the generated product valid? »* is answered by the Variant view.
- *«Which features are interacting with others features? »* is answered by the Context view.
- *«Which features make changes to properties? »* is answered by the Property-Changes view.

The four views can help answer these questions at different levels. Depending on the goal, the view can work with different elements. For example to search which files are implementing a feature, the realization view can be used by displaying, in a list of files, only those which implements a feature. If the developers wants to know which methods, or line of source code are implementing a feature, the realization view takes place

in the view of the file, and only some portions of the source code are displayed. These views work on a decision, is this element part of the view or not? It doesn't indicate whether the whole file or only a few lines are dedicated to a feature.

It is necessary to contrast the advantage of the realization view, as highlighting only one feature with color or displaying only the code of one feature works on the same elements. However, as the realization view only displays the code of the feature, it is separated from the surrounding code. This separation could prevent the developer from understanding correctly or simply understanding how the feature is implemented. For example, if the feature uses a function or a method described somewhere else in the code, then the developer may not be able to understand how the feature is implemented without an access to the complete source code.

However, Heidenreich proposes to apply visualization also on the software models. As Ball et al. [1] says, there are simple representations of softwares, the graph and the hierarchical representations. The graphs are used in most CASE tools. As these tools already provide a representation of the program, adapting the model to the software product line is logical. Heidenreich *et al.* [13] proposed to use this approach on models representing artifacts. For example, the author used it for data and relations models. He also discussed about realization models, which could be in this case the model of the source code.

4.2. Source Code Visualization

As previously mentioned, Kästner *et al.* [21] have developed an annotative approach to produce software product lines. Their approach was based on a IDE called CIDE for Coloured IDE. To associate features with source code, the developers needs to select the portion of source code and use the mouse to choose a feature of the FM. The annotation is then saved in a separate file – containing feature. Their tool is an example of the visualization technique described by Heidenreich *et al.* in section 4.1. They have built two views on two levels.

The two levels are the file and project levels. The file level limits the scope of the view on the currently opened file and uses the nodes of the abstract syntax tree as elements to colour (or hide in some views) in their views. At the project level, the scope is no longer limited to a specific file, but to all the files in the project, the element coloured (or hidden) is the file. These two levels help the developer to understand how features are implemented in the project from a high and low level approach. As Kästner *et al.* [21] say: *«This way the colour metaphor scales from smallest code fragments within a file up to entire directories.»*. See the figure 4.4 for an example of coloured files and source code. The lines of source code are coloured with the colour of the associated feature. The same principle is used for the files in the project explorer.

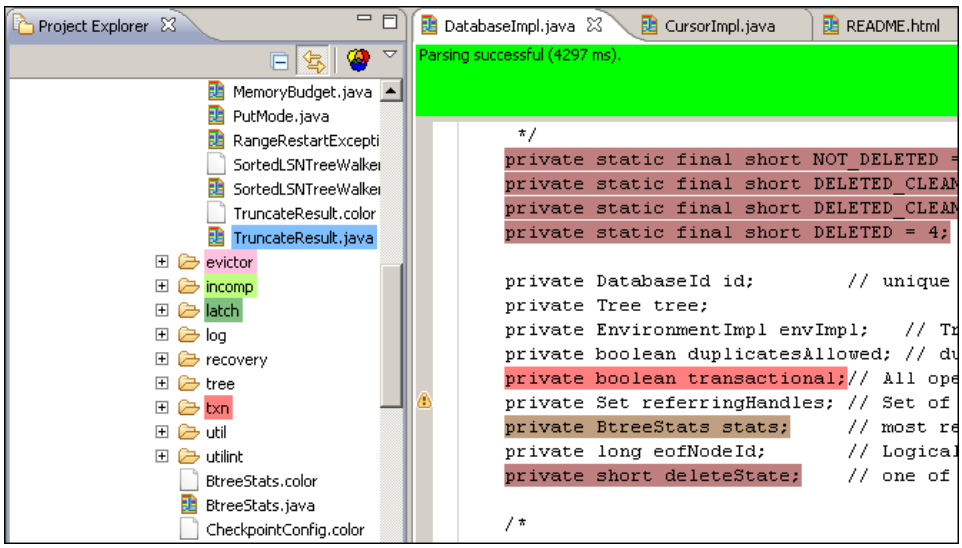


Figure 4.4: CIDE

Both view require the selection of features to modify the views. To do that, Kästner *et al.* [21] propose to use a simple list of features (*« to select features for a view, a simple list of features without further dependencies is sufficient»* [21]*).* The authors believe that it is enough as dependencies are not relevant here, they use instead an indicator to warn the developer if the selected set of features is a valid configuration.

CIDE provides two views. They work on a similar principle to the variant and the realization views described by Heidenreich. The former only keeps the source code that is associated with selected features. As they say, this is close to the result of generating a product. The latter is aimed at tracing a feature. This resolves the issue resolves the issue highlighted earlier: the absence of context if only elements implementing the features are shown. It displays only the code associated with the feature, independently of any annotation for other feature but with some unannotated code that may be relevant to understanding the feature implementation.

However, just this example demonstrates the views given by Heidenreich, it helps understand the limit of these views. While Heidenreich uses this technique on models, they are not limited to any type of model. Therefore, their approach relies on a manual mapping of features to the elements realizing them [13], which could be done automatically in the case of a model of the source code.

Kästner does not provide a solution as, in contract to what is done in a file level, there is no relation between files. Kästner *et al.* just use the hierarchy of files displayed by the IDE. The view inside a file displays the source code annotated by colours and therefore the developer could use the context displayed to understand how a feature was implemented in the program.

Kästner *et al.* [21] also advance the idea of editable views. The developer could use the view discussed previously as a way to edit the source code. As the view would only display relevant code, the developer could focus on the implementation or modification of a set of features. To ensure the developer is aware of the modification he makes, an indication of the annotated source code is still visible. They inform the developer that he is modifying a portion of source code where some source code is hidden. However, the developer is not constrained when he erases a portion of source code which contain a hidden portion, the code not displayed will be deleted too.

Evaluation

The implementation of the views by CIDE, served as an illustration of the advantage of the visualization techniques. It also indicated which views were most important, the realization and the context views (which is discussed in the paper associated with the variant view). However, the study of this implementation also showed the limits of the approach through the absence of relations on a project level view between elements, and the limits of not using a metric for indicating presence of features in files. As this work agrees that there is no need to display the dependencies in the implementation, the developer should refer to the software product line feature diagram. Non trivial software product lines display at least 50 features, which, with a simple list, requires time to find the wanted feature.

The use of colours is also somewhat problematic. As all developer may not be completely able to see all of them and many even see none. If the user lacks of some colours recognition, he may misinterpret the annotation. On top of that, the number of colours that the human eye can perceive is limited. Two colours that are close may be not discerned as different. Therefore the use of colour may be incorrectly interpreted or be missed by persons with troubled vision.. However they are an intuitive way to associate features with source code.

4.3. Integrating Source Code and Meta-Model Visualizations

In *section 4.1* and *4.2*, the visualization techniques were applied to source code and files and elements of the meta-model. However, the visualization techniques do not provide a mechanism to link the different levels of abstraction. This section describes another visualization technique for source code that could help integrate the different levels of

abstraction. This has not been developed specifically for software product lines. Ball *et al.* [1] studied different visualization techniques in the context of software visualization as a whole. Among different visualizations, one kind of visualization is interesting in this context. As stated earlier, there are two levels in the visualization of software product lines: the source code and the project. The source code gives access to source code portions while the project gives access to the file and meta-model. Between both levels, there is a gap that needs to be filled manually by accessing at each file one at a time. The goal of the view is to understand the location of a feature and its relative importance through the use of the display. As this information must be calculated by the developer in this head through out this long process, it is not efficient. By providing a visualization that enable him to see the whole content of a file at a glance, Ball *et al.* offer a way to facilitate the work of the developer. A second visualization will then be used to extend the idea to a set of files.

The idea of the first visualization is to colour elements of the code according to a type. Then by reducing the line to pixels, forming rows of code, the developer can have a preview of how the code is structured. The *figure 4.4* shows an example of source code reduction.

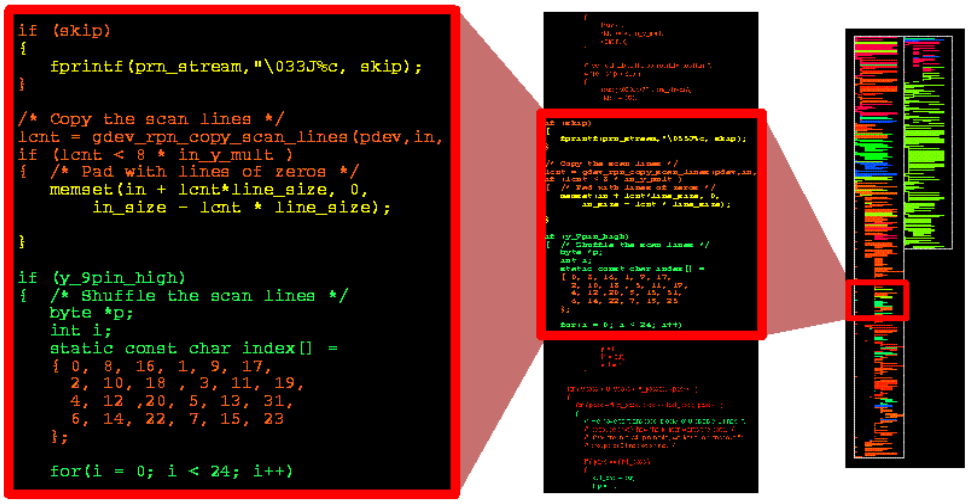


Figure 4.4: Code Reduction [1]

Their approach can be use to represent lines (like in the figure) or to use pixels to represent the source code independently of the look of the line.

In a presentation, Stasko *et al.* [37] explore a visualization, SeeSoft System, that uses the colour and the same principles but at a system level. On top of that they use a zoom mode which enables them to see a more detailed file representation and then accesses the source code. The goal of their view is to help track different operations in the source code, like modifications, bug fixes… The *figure 4.5* shows in the background, every files of a project. When selecting a file, a window appears and zooming to different levels of zooms enables the developer to read the source code.

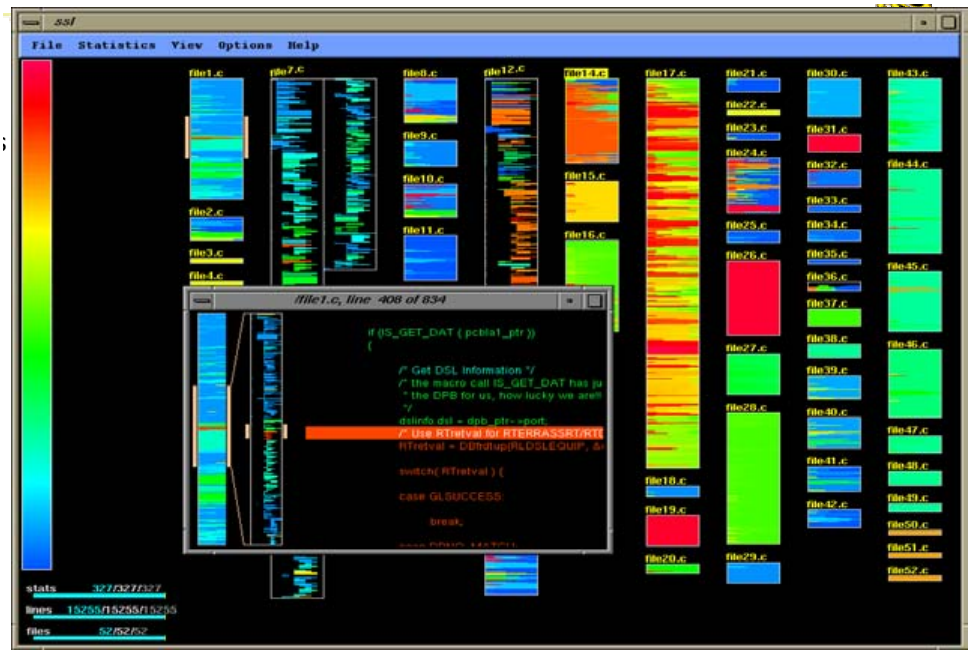


Figure 4.5: Code Reduction with zoom function [37]

4.4. Summary

This chapter highlighted the necessity of visualization for software product lines implementation. It then analyzed a solution proposed by Heidenreich *et al.* [13] and an example of it by Kästner *et al.* [21]. While some of the different views proposed were interesting, the use of colours may be a disadvantage. Finally, the chapter explored two others current visualization techniques used in software modelling.

Part 2. Contribution

This part is the contribution of this work. First, *chapter 5* analyzes the tagging approach. Given its limitations, a set of goals and requirements for a tool support is presented. *Chapter 6* describes the design choices and the implementation of the tool, XToF. A guided tour of XToF is given in *chapter 7* and is followed by an example illustrating the tool in *chapter 8*. In *chapter 9*, the future work for the tool is described. Finally, *chapter 10* concludes this work.

5. Tagging Approach Evaluation

The previous part provided an overview of the different approaches available to implement SPL. The first section of this chapter will describe why the tagging approach is the lightest weight approach by comparing it to other annotative approaches. As stated in *section 2.3.3*, the annotative approaches have some disadvantages that can be improved by using a tool as a support to these approaches. The second section describes these disadvantages. *Section 5.3* describes the requirements of a such tool and *section 5.4* contains examples of use.

5.1. Comparing Tagging Approaches to other Annotative Approaches

Each approach has been described and analyzed in section 3.1. This section will compare them to the tagging approach to justify why the tagging approach is the most adequate appropriate for our light weight goal.

5.1.1. IFDEF

The #IFDEF approach uses begin and end markers associated with a parameter to annotate portions of source code. It can be used for portability [36].

As the developer is free to insert markers, he may miss mark the code. He can annotate a beginning bracket and close the annotation before the closing bracket. The generated code is then invalid. The tagging approach use nodes of the abstract syntax tree, therefore enclosing brackets are annotated correctly. Secondly, the markers used are integrated in the logic of the program[1], increasing the complexity in reading the code and in understanding it. The tagging approach uses annotations which are separated from the text of the program being simply commentaries.

5.1.2. Frames

Frames use tags and annotations to mark portions of source code that need to be transformed into modules [26]. Frames uses annotations to indicate which portion of source code is a module. It also uses the annotations to indicate the rules for combining the modules. In contrast, the tagging approach separates the annotations for identifying features source code from the rules saved in the FM. Separating enables the developer to design and structure the abstractions of features differently from their implementation [7].

5.1.3. CIDE

CIDE is an IDE which uses colour to indicate features in source code Kästner *et al.* [21] claim that saving the annotation in a separate files avoids modifying the text of the program. However, as with the tagging approach, using comments to save the annotation achieves the same goal. The main issue with their approach is that it requires the use of a specific or adapted IDE. On top of that, it is impossible to modify the source code outside this tool without losing all the annotations, unlike the tagging approach.

CIDE enforces rules to provide a syntactically safe annotation. The tagging approach, by construction, limits annotation to nodes of the AST, providing a syntactically non-arbitrary annotation. However, CIDE goes further and allows only annotation of mandatory nodes (mandatory nodes are nodes that cannot be removed without syntax errors). To annotate features, CIDE requires the developer to use a specific function, while the tagging approach allows him to just type the tags.

One issue with using annotation is, as in documenting source code, the annotation may be incomplete. It is common to see methods in a program that don't have any comments or specifications. In CIDE and the tagging approach, as that the annotations are

[1] #IFDEF markers can be considered as being instructions like if-then-else clauses.

used to generate the products and the annotated source code is compiled several times, this ensures that the software is completely annotated [21].

Both approaches avoid obfuscating the code. The tagging approach uses tags inside comments and CIDE entirely relies on colours. In both cases, the text of the program is not modified. However, if the developer has some issues in identifying the colour, he may miss some information. If no features are selected for highlight, then the developer is unaware of the portions of source code annotated with features. The tagging approach does not always show the scope of the annotation but always indicates the presence of an annotation. As the CIDE approach relies on a tool, it also provides support to annotate and view annotation. The tagging approach requires a tool only to prune. However, a tool support for this approach could be useful and will be described later.

5.1.4. Summary

The tagging approach relies on ideas close to the CIDE approach, however it removes the necessity of a specific tool reducing the weight of this approach and enabling multiple uses of the tagging approach. Therefore, the tagging approach is a light weight approach to implementing software product lines.

The comparison between the three approaches highlights the necessity of having a tool for support. Even if the tagging can be done independently of any tool, the possibility of seeing the scope, using a feature mode and navigating through features are important functions that could support the approach.

5.2. A Tool to Support the Approach

In the section 2.3, it was established that annotative approaches had some flaws compared to the compositional approach. These are the modularity, traceability and the safety and could be solved by use of a tool support. When comparing the tagging approach to other annotative approaches, the main issue with the tagging approach is the lack of clarity with respect to the the scope of a tagging.

Modularity is the ability of the developer to use a feature in an other context. For example in compositional approaches, a feature is developed into a module and this module can, in some conditions, be used in an other context. Modularity is out of the scope of this thesis and therefore out of the tool built to support the tagging approach.

As the compositional approaches use separated modules to realize features, the traceability is immediate. One feature is in one module and no other is in it. In the annotative approach, the advantage of not modifying the software for the approach also renders the task of tracing a feature more difficult. It is scattered through the software and becomes a crosscutting concern. However, the tags contain the trace of features implementation.

Therefore, the developer with a search and find function could locate every occurrence of a feature. However, a tool specifically built for that would be more useful. By using the feature model, a developer can select a feature, the tool then displays a list of every location of portion of source code associated with that feature. Then the tool could show the scope of the tags so that the developer knows exactly which part of the source code in each files is dedicated to the selected feature.

As in the compositional approach a module is syntactically safe. However in the case of an annotative approach, rules are needed to avoid arbitrary annotation. While this approach only enables the developer to annotate nodes of the abstract syntax tree, it also allows him to annotate mandatory nodes (mandatory nodes are nodes of the AST that cannot be removed without syntax errors). Therefore a tool may be used to enforce this rule. There is also a second possibility to help the developer, the use of the minimal set of features can be used to ensure the safety.

Tagging the source code with features requires using the exact name of the features. It makes the task of annotating more difficult as the developer has to remember the feature name. The developer could also make errors in the name and the tags would be ignored generating an erroneous product. A tool, which could check whether the name

used are correct or not and propose names of features to the developer while he is tagging, would reduce the weight of this approach.

The tool should rely on the feature model, display a feature diagram to help the developer remember the location of the feature. When proposing features names to the developer, the tool should enable both use of short name and of the beginning of the path to propose the long name of the feature.

5.3. Requirements

From the experiments carried out by Boucher *et al.* [3] a list of necessary functions for a tool was developed on top. Some requirements were added later as early tests showed their necessity. The *table 1* lists them and for each requirement, the associated priority is indicated by an X. In the rest of this work, *R1* to *R14* will be used to indicate the requirements. The table is repeated in *annex D* to be used during the rest of the work.

R	Function\Priority	High	Medium	Low
1	Load an FM	X		
2	Display an FD	X		
3	Create an FM			X
4	Checks Tags	X		
5	Auto-completion for Tags		X	
6	Drag-and-drop to tag			X
7	Display scope	X		
8	Display list of tagging location			
9	Navigate		X	
10	Configuration	X		
11	Retain state of a configuration		X	
12	Prune	X		
13	Minimal set of features		X	
14	Project Level visualization		X	

Table 5.1: Requirements and their priority

1. *The tool should provide a mechanism to load a feature model.* Different functions of the tool require the developer to know the name of features, such as checking the tags, pruning, auto-completion, etc… The function configuring and minimal configuration require an FM. As many function depends on it, it is assigned a high priority.
2. *The tool should display the feature diagram associated to the SPL project.* Displaying an FD helps the developer remember the name of the features and how they are structured. It can also serve as an entry point for functions associated with features such as configuring and lists of tagging. It has a high priority as it is a necessity for using the tool and helping the developer.
3. *The tool should enable the developer to create an FM.* The tool already provides a mechanism to load a feature model. A mechanism to create and modify an FM inside the tool would prevent the developer from having to switch to another tool. Each modification to the FM should be echoed in the tagging to maintain coherence. As the create function is already sufficient for every others requirements and modifying FM is not a necessity, it has a low priority.

4. *The tool should check tags correspond to existing features.* The tool ensures that every tagging only contains existing features. The tags are compared to the names of the features contained in the FM. If a feature name is incorrect, empty or unknown, the developer is warned. As this function can prevent the developer from errors due to the tagging, it has a high priority.
5. *The tool should provide an auto-completion for tags.* While tagging, the developer can call for an auto-completion function. The tool should propose names and paths of features to help him choose the correct feature name with the characters already written. This accelerates tagging by reducing the need to write the long names. This has a medium priority, as tag checking can already provide a way to avoid typing errors, but accelerates the tagging.
6. *The tool should provide a drag-and-drop mechanism to tag source code.* The drag-and-drop could be used to quickly tag a portion of source code. When hovering the source code with a feature, the portion that would be associated is highlighted. This has a low priority as it is a redundant function compared to auto-completion.
7. *The tool should display the scope of a tagging.* The whole tagging approach relies on how a tag is associated with a portion of source code. Developers may not be aware of how the AST is structured and how the association is done. So, the tool should display the scope of a tagging to let the developer check if it is the intended association. As this feature is highly important, it has a high priority.
8. *The tool should display a list of every tagging.* One of the disadvantages of annotative approaches is traceability. So, the tool should display a list of all tagging. Selecting a set of features should reduce the list displayed to tagging containing at least one of the features. As it is one of the disadvantages of the annotative approaches, it is classed as high priority.
9. *The tool should provide a mechanism to navigate through the tagging.* The tool should enable the developer to go to the location selected in a list of tagging. As it is also part of the traceability, it has a high priority.
10. *The tool should provide a mechanism to configure feature models.* As the pruning requires a configuration to prune an artifact, the tool should provide a function to realize configurations. While a configuration is simply a set of selected features, the complexity comes from the legal criteria of the configuration. The tool should also provide a propagation mechanism to accelerate configuration. Each time a feature is selected or unselected, given the selected features and the rules of the feature diagram, features that are required are automatically selected and the features that cannot be selected are unselected. The tool hides the complexity of feature model constraints and accelerates the configuration by restraining the number of choices to valid ones and by automatically making mandatory choices.
11. *The tool should save the current state of a tagging.* The tool should propose a function to save the current state of a configuration and restore it later. This would enable the developer to have a set of configurations ready to prune or to complete before prune. This would be an important function in a test procedure by configuring and pruning several configurations. This function is classed medium priority as the developer can use the tagging approach without it. However it could greatly reduce time needed to configure.
12. *The tool should provide a pruning mechanism based on a configuration.* The tool should provide a pruning mechanism using a configuration. As this is part of the tagging approach, it has a high priority.
13. *The tool should provide a mechanism to compute minimal sets of features associated with a feature.* When using the tagging approach, some types of errors can appear. Using in a feature, a type, function or variable that is declared in association with a set of features can sometimes lead to an error as the target is pruned. To prevent such errors, the developers can use a design rule: *«Each feature can only use variables, functions and types declared by itself or by its principal dependencies. »* [3]. The minimal set of features is the set of features that will always be present with a selected features. It has been classed at a medium priority.
14. *The tool should provide a visualization technique at a project level.* The different mechanisms and functions present in the tool only provide visualization at a file level. The visualization should help trace features implementation at a project level.

5.4. Scenario

5.4.1. Pre Requisite

XToF requires a Feature Model for displaying a feature diagram and for any operations. The feature model is stored in a XML file compatible with SPLOT / SPLAR file model (as XToF does not intend to provide creations of Feature Model in the first version, the SPLOT website can provide a Feature Model creator). XToF will provide an explorer to select a file representing a model, it will be copied inside the project and then used as the feature model associated to the current project. For each process, there will be a list of sub-processes that the tool should carry out.

5.4.2. Implementation

This is the classical process from the software development. The tool should provide a help for maintaining traceability and delimiting features insides the code. Tags are used for both purposes. They contain the features used to tag this waypoint. During the coding, a developer can use the auto-completion to tag a block of code. He can refer to the displayed feature diagram for more information about the features available. A line of code that is not associated with any waypoint will be part of every generated product. The developer should also be able to review which blocks of code are associated with a set of features.

1. Associate a set of features with a block of code.
2. Review code associated with a set of feature and inversely.
3. Provide help for the developer when tagging blocks of code.

5.4.3. Verification with the Minimal Set of Feature

To help the developer in coding, the tool provides several mechanisms. Every mechanism to display the scope and list the tagging associated with a feature are part of this mechanism but also the minimal set of features. Considering the feature model, the tool should generate a minimal set of features associated with a selected feature. The developer can than use that set of features to generate a product and checks its validity. The code of the generated product should simply be compiled to perform the verification. By applying this test to each feature of the feature model, the developer ensures the code is correctly tagged.

1. Display a list of every minimal set of features (i.e. display every features).
2. Select one and generate the related minimal product.
3. Provide a project associated to the generated product to help the user with the compilation.

5.4.4. Product Generation

Once the process of coding is finished, the developer will use XToF to generate products or while he develops, to test the tagging. A product is a program generated from a set of features. A valid product is a product whose set of features respects the model constraints. The generation of a product prunes the code related to the set of unselected features from the product.The developer configures the feature model by selecting or unselecting features successively. At each stage of the configuration, a sub set of features can be automatically configured according to the constraints of the model. For example, in an XOR group only one feature can be selected, once the user chooses one feature of the group, the sibling features cannot also be selected and by propagation, they are unselected. The developer should be able to see the impact of his actions through the modification of the feature diagram.

1. Provide an environment of step by step configuration.
2. Reduce the number of steps needed by propagating choice and adapting the display.
3. Provide history of choices and a cancel option.
4. Inform when the configuration is complete.

5. Generate the pruned code from the set of features in a folder by using a
    a. Built-in pruner.
    b. Pruner provided by the user.

5.5. Summary

The tagging approach relies on features names. It associates features to nodes of the abstract syntax tree through their tags. This annotation is tool independent. However it could be supported by a tool to facilitate the work of the developer. Using the annotation, the pruning removes the code unused for a configuration and generates a pruned source code that can be compiled to manufacture a specified product.

This approach through the annotative approach and the use of tags, reduces the amount of modifications that need to be made to the software and to its development. In summary, the tool with its support to the processes, further reduces the difficulty of using our approach to implementing software product lines. Therefore, it is a complete and light weight approach to implementing software product lines.

6.Design of a tool Support

Chapter 5 established the necessity of a tool to support the tagging approach and defined the requirements it should answer to. Therefore, the idea of building a tool emerged. This tool can be used to put into practice the tagging approach described in the previous chapter.

This chapter will first give a short description of the tool developed (XToF), then describes the design solution and finally its architecture.

6.1. What is it?

XToF (Cross(X) Tagging of Feature) is the tool developed to support the tagging approach. It was designed using the requirements of *section 5.3*. It is an Eclipse plugin. Eclipse is a well known IDE used for Java, C, PHP, etc. It is open-source. XToF integrates within Eclipse to provide support. In the current version, it supports Java and C languages to implement software product lines. However, it is possible to use others languages by defining an extension for XToF. XToF has been published under the form of a tool demo paper by Gauthier *et al.* [12] in the workshop VaMoS 2010[1].

It has implemented the requirements 1,2, 4,5, 7-14 of *section 5.3*. As a reminder, see *table 6.1*. In this work *R1* to *R14* are used to reference a requirement. The table is repeated in *annex D.*

R	Function\Priority	Implemented
1	Load an FM	Yes
2	Display an FD	Yes
3	Create an FM	No
4	Checks Tags	Yes
5	Auto-completion for Tags	Yes
6	Drag-and-drop to tag	No
7	Display scope	Yes
8	Display list of tagging location	Yes
9	Navigate	Yes
10	Configuration	Yes
11	Retain state of a configuration	Yes
12	Prune	Yes
13	Minimal set of features	Yes
14	Project Level visualization	Yes

Table 6.1: Table of requirements

6.2. Implementation Choices

During the development of XToF, several implementations decisions had to be made. They are explained in the next subsections.

[1] http://www.vamos-workshop.net/

6.2.1. Selected Backends

To accelerate the development of the tool, a search for existing tools and solutions that could be re-used started. Two backends were needed, one for handling tags, the other for managing FM. The two solutions selected were respectively TagSEA and SPLAR.

TagSEA

XToF needed to parse files to search for tagging, then parse the tagging for each feature. XToF should also maintain data about every tagging and their location. TagSEA appeared as an adequate tool for such functions.

TagSEA[2] is a tool described by Ryall [34] and developed with the help of Del Myers. It originated from a research collaboration between University of Victoria's *Computer Human Interaction & Software Engineering Lab*[3] and the IBM Watson Research Centre[4]. It is aimed at *«providing more structure for organizing annotations by allowing developers to define their own vocabulary, and allowing developers to link crosscutting concerns in the software»* [34].

TagSEA enables the developer to tag the source code for locations of interest. TagSEA uses the idea of tags in the social context. The user is free to use his own vocabulary to add information to the source code. He can also add some metadata to the tags, like the author and the date. The tags do not only contain keywords and metadata but also a message. All this information is then displayed in TagSEA.

TagSEA uses the metaphor of waypoints to help navigation in the source code [41]. The waypoint is a position that is marked as interesting in navigation systems. By combining the waypoints, a route is formed, a way of navigating through the code. The developer can also make presentation of code using the routes.

TagSEA displays different views of the tags. One is a list of tags. As tags can be hierarchical, it helps organizing them. When a set of tags is selected, every locations is displayed with information such as the file name, the associated element of source code, the date, the author and the message. This helps the developer in searching for a specific location.

On top of displaying this informations and the tags, TagSEA enables the developer to modify them. The information associated with a specific location can be changed and the corresponding tagging will be updated too. TagSEA also enables the tags themselves to be renamed and deleted. Therefore it helps the developer in the task of maintaining the tags and avoids using a search and find function. TagSEA also displays tags under the form of a tag cloud. The size and importance of the tags in the representation are linked to their usage.

TagSEA support out of the box the C(++) and Java languages. However, the tagging is not limited to source code as files, breakpoints and URL can be tagged. TagSEA also enables developers to define their own kind of tags and points of interest. This is realized by using the plugin mechanism brought by Eclipse, the IDE in which TagSEA is implemented.

As TagSEA is used, every function that TagSEA provides to the user can be used for the benefit of the tagging approach. TagSEA can be used as a backend for the tags parsing and management and as a front-end to access features, tags and locations. The next list describes the reasons behind choosing TagSEA as a backend for tagging.

- TagSEA is aimed at *tagging source code*, it also uses comments to embed tag in the source code. It provides tools to list tags, their locations and accesses the file and lines where they are written. It is very close to this approach in term of how tagging is realized.
- TagSEA is an *Eclipse Plugin* and using it may limit the tool to this specific IDE. However, Eclipse is a mature IDE used by many developers and provides frameworks for

[2] http://tagsea.sourceforge.net/

[3] http://www.thechiselgroup.org/

[4] http://domino.research.ibm.com/cambridge/research.nsf/pages/index.html

developing plugins, working on source code, files and projects, creating views and displays. Therefore, it would reduce the time needed to develop the tool.

- TagSEA uses the plugin mechanism of Eclipse for working, but it also enables the *extension* of the languages and types of tags supported by TagSEA. Therefore, it would reduce the time needed to implement the tags. As Eclipse provides mechanisms of extension by using plugins, it enables the tool to have the capacity to be easily extensible for other languages. As Eclipse enables the plugin developer to use the functions provided by the IDE to support development, the tool could re-use the functions and mechanisms known. Therefore, developers wouldn't face unknown mechanisms.
- TagSEA served as a *tool for research*. It has been validated in the context of tagging source code for points of interest and has even been used by a developer to locate crosscutting concerns [34].
- TagSEA uses tags to help the developer to remember and re-find locations of interests. In an other work, TagSEA was also described as a *navigation* mechanism in the source code by Storey et al. [41]. The tagging approach would benefit from such mechanisms to facilitate the work of the developer.
- The main developer, Del Myers, who worked on TagSEA could be contacted and was ready to provide some insight of TagSEA to help develop XToF.

SPLAR

The requirements (of section 5.3) *1* and *2* (load an FM and display an FD) imposed using an FM. Others requirements imposed realization of a configuration. FM are complex (relation between nodes, additional constraints…), computations are done through specific tools such as SAT Solvers. As the purpose of this master thesis is not to develop a tool for reasoning on feature models, to make computations on an FM, the FM is transformed into formulas solved by a SAT solver. This mechanism is used when doing a configuration and it serves to propagate the (un)selection of features and to ensure only valid configuration.

Mendonça has realized tools to make computations on feature models that are suitable. Mendonça do research on large scale reasoning techniques for feature models using SAT solvers [27]. He produced two tools, SPLAR a reasoning library and SPLOT [28] an online tool using SPLAR as a backend. The following list describes why the tools developed were selected as a backend for managing FM.

- The different tools were conceived to provide an efficient reasoning for large scale FM. The methods used in the tool have been published and validated.
- The feature model is saved under a specific language. It is aimed at producing compact notation while remaining legible and modifiable by humans.
- Two tools were available and offered different integration solutions possible. SPLAR is a Java library that can be integrated in Eclipse and SPLOT is a website that could be accessed using the Eclipse internal browser.
- The tools propose creation and modification mechanisms for SPL FM. Although it was out of the scope of this work, the tools could be used in a future work to extend XToF. As the current version didn't provide any creation mechanisms, the developer can use SPLOT or create manually a feature model for XToF.
- Mendonça agreed to help in integrating his solution in XToF. This provides an advantage to quickly integrate feature model reasoning in the tool.

There were two tools available, SPLAR and SPLOT and three solutions to integrate an FM backend (SPLOT could be integrated in two different ways.

1. First solution is to use the browser integrated in Eclipse to access the SPLOT web site, the developer works online and XToF loads the resulting file from the website. This solution requires a constant Internet connection to effect any operation on the feature model. The tool would have to re-interpret the file describing the feature model, reducing the benefit of having a complete SPL backend. The computer of the developer is released from the computation on feature diagrams.
2. Instead of using the website directly, an XML interface is used with questions and answers to communicate with SPLOT. As the website is not used directly, the tasks

can be integrated into Eclipse. This solution also requires a constant Internet connection. The computer of the developer is released from the computation on feature diagrams. Simple questions and answers are used to describe the feature diagram. Therefore, there is no need to parse and reinterpret a file. The developer of SPLOT wanted to add an XML interface to SPLOT in the short term.

3. The last solution is to use SPLAR. SPLAR is a Java library used in SPLOT. XToF would use the library to make the computations. Integrating SPLAR requires an understanding of how it works and how it has been designed. As the configuration system used in SPLOT is not available, it needs to be redeveloped for XToF. However as XToF already includes a feature diagram display, it can be adapted to enable configuration. This solution permits the complete integration of the backend. Therefore, the user is unaware of the underlying mechanism. It also enables the possibility of using a different tool if needed. There is no Internet connection required. Though XToF does not provide creation functions, the feature model comes from a file that is interpreted by SPLAR and not by XToF.

The solution chosen is SPLAR as SPLOT solutions required an Internet connection for all reasoning done on feature models. The website may be not maintained in the long term. SPLAR provides an permanently available solution which would be fully integrated in the Eclipse IDE.

6.2.2. Solving the Scope of a Tagging

The authors of the tagging approach rely on the AST to provide a syntactically pruned code [3]. They state that the functional block of the AST is associated. However, they don't give a precise rule for associating nodes of the AST with a tagging. To provide a tool independent approach, the rules of association must be fixed so that any tool using the tagging has the same association for the tagging.

The scope of a tag is explained in the case of the Java language. However the same principles are used for others languages such as the C language. The scope of a tag is the node called the *portion node.* To find the portion node, three special nodes are required: the *tag node*, the comments containing the tagging, the *following node* and the *enclosing node*, the node that contains directly the *tag node* as displayed in *figure 6.1*. The *following node* is used to check if the following node is a sibling of the tag node. To understand what the following node, it is necessary to understand how the research is done. The AST is flattened on the character axis, see *figure 6.2*. The reason behind flattening the AST is due to the absence of direct access to the AST. Eclipse provides a visitor pattern to access the AST. On the character axes, a node is defined by a range: its beginning and its end. Using that notion, the *following node* can be defined as the next node on the character axis. If the *following node* is in the *enclosing node*, then it is the *portion node*. This is the case in the figure. All the computations are done using the characters axis. The corresponding source code is displayed with the node designated in the *figure 6.5.*

The enclosing node is used to determine if the following node is a sibling of the tag node. If the following node is not inside the enclosing node, see *figure 6.3* for the AST and *figure 6.4* for the character axis, then the following node is not the portion node. In this case, there is no source code associated with the tagging.

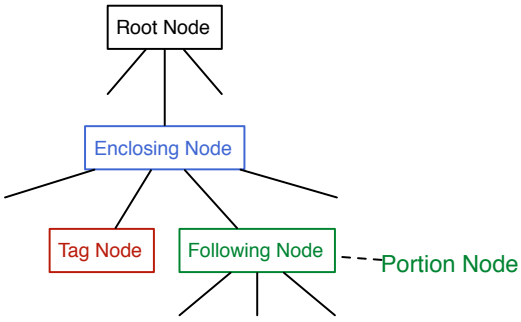


Figure 6.1: AST with the following node being the portion node

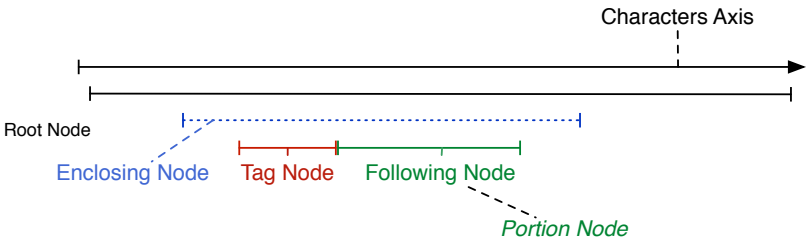


Figure 6.2: Representation of an AST on the characters axis with the following node being the portion node.

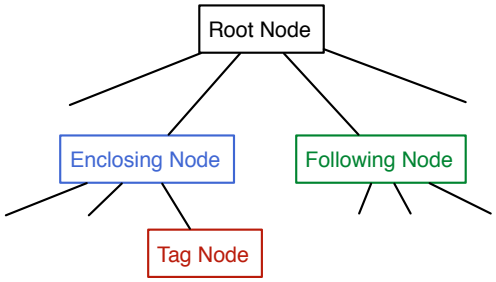


Figure 6.3: AST with no portion node

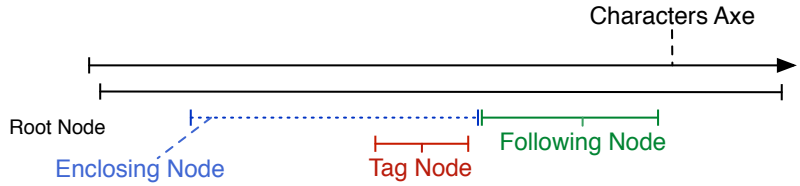


Figure 6.4: Representation of an AST on the characters axis with no portion node

This rule always works except for two cases: the case when there is JavaDoc declared and with Switch-Case statements. For the first case, when a method is declared, if any JavaDoc is present, it is part of the method declaration. If the tagging is declared between the JavaDoc and the beginning of the actual method declaration (see *figure 6.5*), the following node is the *modifier node* (if present or the next one). The method declaration becomes the However, if there is no JavaDoc, the same tagging will be associated with the whole method.

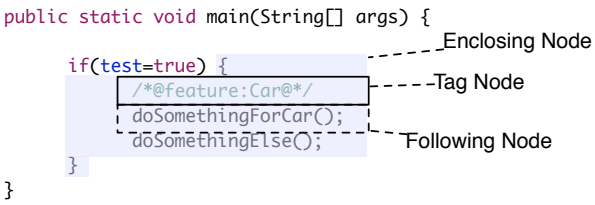


Figure 6.5: Association of tagging

Therefore, we needed to develop a solution to keep a coherent association, even if any JavaDoc is present. When a node declaring a method is considered during the research, its start is set at the start of the range of the *modifier node* (or the next present node) see *figure 6.6*. Therefore if any JavaDoc is added to the method declaration, it does not block the association.

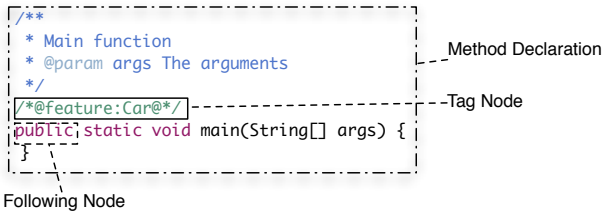


Figure 6.6: Association in the case of JavaDoc

In a switch-case statement, the case node is limited to the declaration of the case and each of the instructions included in the case are sibling nodes of the case. Therefore the association method would always limit the association to the case declaration only (see *figure 6.7*). The case can be compared to the *then* statement of an *if-then-else*. By putting a tagging before the *then*, the developer expects the whole *then* statement to be associated. It should be the same in the *switch-case* to keep it coherent. Otherwise, the developer would have to tag each instruction (see *figure 6.8*), including the break if present. The decision that was taken to keep it coherent is to artificially treat the instructions of the case as one node.

```
switch(i) {
/*@feature:Car@*/
case 0:
      doSomething();
      break;
case 1:
      doSomethingElse();
      break;
}
```

Figure 6.7: No specific rule for association

```
switch(i) {
/*@feature:Car@*/
case 0:
      /*@feature:Car@*/
      doSomething();
      /*@feature:Car@*/
      break;
case 1:
      doSomethingElse();
      break;
}
```

Figure 6.8: Manual tagging of each instruction

However in this strategy, which nodes are included? In a switch-case, the instructions of the next *case* could be executed too if there is no *break* statement. Two choices were available then; to use this approach for the tagging (see *figure 6.9*) or limit the instructions to the break included or before the next case (see *figure 6.10*).

```
switch(i) {
/*@feature:Car@*/
case 0:
      doSomething();
case 1:
      doSomethingElse();
      break;
case 2:
      doNothing();
      break;
}
```

Figure 6.9: Association until the next break

```
switch(i) {
/*@feature:Car@*/
case 0:
      doSomething();
case 1:
      doSomethingElse();
      break;
}
```

Figure 6.10: Association until case or with break

The first solution offered is not suitable in some context as the resulting pruned code would removed the case included in the association (see *figure 6.11*). The second case is rejected too. The developer cannot remove only the first case without affecting the second one. Therefore, the scope of the tag includes all the nodes following the case node until a break node, included in the scope, or an other case node, not included in the scope. This choice enable the removal of only the directly annotated case as in *figure 6.12.*

```
switch(i) {
case 2:
      doNothing();
      break;
}
```

Figure 6.11: Pruned code with association until the next break

```
switch(i) {
case 1:
      doSomethingElse();
      break;
}
```

Figure 6.12: Pruned code with association until case or with break

6.2.3.From Feature Name to Tags

The syntax of the tags was illustrated in the example with the description of the tagging approach in section 3.9. Then in section 6.4, the scope association was described. In the tagging, the tags are the name of the feature. However, what happens if there are two features that have the same name, but not the same parent. Is this allowed by the FM language? How to deal in the tool given these possibilities? What is the rule for naming the features and the associated tags?

Several options are possible. First, impose a different name on every feature and use only the name of the feature. Second, allow features to have the same name and use as tags the name of feature preceded by the name of the parent if a feature name is not unique. Third option, still allow features to have the same name and use the path from the root and the feature name as tags. The next paragraphs analyses the three options.

First, we analyzed the solution imposing the unique name of features. The main advantage of a such solution is that tags are short, they only contain the name of the feature. Therefore the taggings are easily readable. However, this solution has several disadvantages. It imposes a different feature name on every feature in the feature model used. As the tagging approach is intended to be tool independent, it should not make assumptions or impose conditions on how features are named. Using only unique names may lead the developer to design an FM with features name like *lh_thumb* for the thumb of the left hand. On top of that, having only the name of feature used make for a more complex understanding of where the feature is located in the FD and what is its purpose.

The second approach uses only the name of the feature if this name is unique and if not, imposes the use of the name of the parent feature in the tag. For example, if there are two features named *thumb* in two different parents, *left_hand* and *right_hand*, the corresponding tags are *left_hand.thumb* and *right_hand.thumb*. This solution still offers short tags (a bit longer than the first but only in some cases) and the same disadvantage as the first solution. However a supplementary issue appears. If two features can have the same name, why could their parents not too? Instead of talking about *left_hand* and

right_hand, the *body* could be first split into *left_part* and *right_part* then each has a *hand* feature. The corresponding tags would be *left_part.hand.thumb* and *right_part.hand.thumb*. Then the rule to impose the parent name becomes complex and could in some case result in a long list of parent names before the actual name of the feature. This solution lacks the simplicity of the first and in some cases could be as long as the third.

As we talk about the third, imposing the path from the root to the feature name included is always the same. There is no need to know if the parent has a unique name and so on. The tags are however always long and provide a less readable tagging. For example in the two examples in the previous paragraph, the tags for the thumbs would be *body.right_hand.thumb, body.left_hand.thumb, body.right_part.hand.thumb* and *body.left_part.hand.thumb*. The tags are long but it is easy to remember that the thumb of the third tags is attached to a body, in the left part and is not attached to the right foot. This solution provides meaningful tags that can be easily traced in the FD. The choice of imposing the root name is discussed in next paragraph. Concerning the root name, it is imposed as a way to provide a possibility to use several FM in the same project. For example, some research by Czarnecki *et al.* [9] proposes to use a split FM to facilitate the configuration. In large projects, it may be useful to apply the same idea to the implementation itself.

As the second solution is simply a more complex mix between the first and third approach, it could not be chosen. The choice was restrained to the first and third solutions. As the first solution imposes a strong requirement and may lead to complex names, it was eliminated and the third solution was selected. It provides a simple rule to form tags that could provide meaningful tags as the hierarchical tags of TagSEA [34]. It should be noted that spaces are not allowed in the tagging and therefore spaces in feature names are replaced by an underscore.

6.2.4. Display a Feature Diagram

The requirement *2* of *section 5.*3, states that the tool should display an FD. The FD is also used to access most functions related to features. However in CIDE, they relied on a simple list. Why should XToF use an FD instead of a simple list?

Displaying a list is longer than an FD where some nodes can be closed. Therefore when the developer is working, it is easier for him to find a feature he needs. As in *section 6.2.3*, the choice was made to use path of features as tags. Therefore displaying an FD at any time and using it as way to access functions helps the developer remember the paths of the features. It makes the approach coherent.

6.2.5. Project Level Visualization Techniques

One of the requirements is to provide a project level visualization to help in traceability. If there is a project level, it has also a lower level. The different visualizations techniques had to be separated into these two categories. The level of abstraction was used to define them. Before discussing the project visualization, the lower level should be explained. This is the file level and aims at providing visualizations that help the developer understand how the features are implemented in a file or a set of files. Different mechanisms are used and are explained in several sections of the *chapter 7*. They are the *scope highlighting* (displays the associated source code of a tagging), the *associated feature* (displays the feature associated to a line of source code), the *file labelling* (a label precedes file names that contains selected features), *filters* (hides files that do not contain any selected feature) and *file selection* (displays the feature contained in the selected files).

This mechanisms are useful to analyze how a few features are implemented in a few files. However, if the number of features selected is growing, then more and more files are displayed, until every file may be displayed, and vice-versa. The project visualization also aims at providing information to the developer of how the features are implemented. It is designed to fit every features and every files of the project.

6.3. Architecture

This section provides first a description of the architecture of XToF, then describes the architecture of TagSEA.

6.3.1. XToF Design

The diagram of component (*figure 6.13*) describes the architecture of XToF and will be completed by a more complete description in *section 6.3.1*. There are five components. The core component is *VFD* which contains the FM and can be used to access others functions. The component *Parsing* provides highlight and the parsing and pruning independent of any language, while the actual parsing is realized by *Java Parsing* (or *C Parsing*, not represented, for C). When the selection is modified, the *Filters* can ask to be notified. *Filters* and *Project Visualization* provide visualization techniques using the information about tagging of *Parsing*. The next paragraphs will describe why this design was selected.

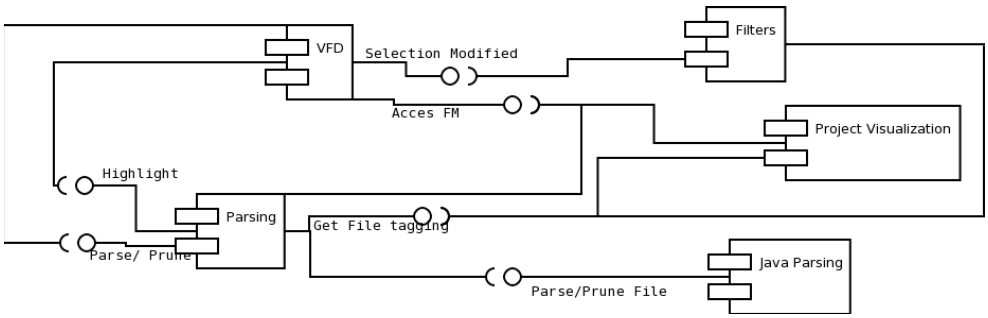


Figure 6.13: Diagram of component

When designing XToF, choices had to be done. They are described here and why they have been selected. As TagSEA was chosen for, amongst many reasons, its compatibility with our tagging, its implementation in Eclipse, see *section 6.2.1* for more details, it imposed the choice of Eclipse. Therefore, the implementation of XToF had to been done as an Eclipse Plugin. An Eclipse plugin can be composed of several plugins. Each plugin can describe extensions points, others plugins can use them to extend the functions of the plugin. It can also provide an API. TagSEA can be extended to new languages and tagging by using extension points. A plugin using this extension, must provide some informations and can re-use the scanners TagSEA provides. The plugin then only has to scan each tagging that TagSEA finds and sends back the tags found. It does not have to handle anything else, not even the data persistence.

XToF provides many different functions. As implementing it in one single plugin would be complex and difficultly maintainable, these functions should be regrouped into separated plugins. How many plugin should it have, how to regroup their functions? First, the reflexion can be done on the core function of XToF. The main function of XToF is the SPL and the FM. Loading a feature model and handling the request (configuring included) to the FM are the main functions. The others functions can be regrouped into three coherent categories: the parsing of tagging, the handling of an FM (as a point of access to functions too), traceability mechanisms and project level visualization. Regrouping the functions into these categories provides coherency to each plugin.

As the FM is the core of XToF, should the FD display be included? Displaying the FD in a separate plugin would allow the possibility of using a different type of FD display or more than one. However, as there would be many accesses to the core for information on the FM and as it will serve as the main point of access to functions, it should be regrouped with the core.

Concerning the functions that are specific to languages, like the parsing of tagging and the traceability mechanisms, there should be one for each language. To reduce the amount of work needed to extend XToF to new functions, a plugin should provide classes that could be extended and tools ready to re-use. This creates a plugin that does not do any parsing but furnishes tools. Parsing of tagging is close to scope highlighting

and can regrouped. Traceability mechanisms are the filters and labelling of file names. They could be integrated with the parsing plugin for the same language, but as functions are distinct, providing a different plugin maintains a higher coherency in each plugin. The project visualization is intended to be language independent, for the first version at least. Therefore, it is not regrouped with the two previous plugins but rather as a new plugin.

Originally the plugin was called VFD as visual feature diagram. Therefore the name was kept for the view name and the artifacts names.

Plugins

- VFD
- Parsing
- Java/C
- Filters Java/C
- HighLevelView (Concern/Zest)

The two next graphs represent the dependency between plugins. The second is a simplified schema. To make the description simpler, only the types of plugins are represented. For example Java/C are two separate plugins but as they are independent and provide the same functions but for a different language, they are grouped.

VFD is the core plugin. Parsing defines the parsing of tags at a high level and Java/C does it at a low level. Filters Java/C is called by parsed in case of modifications to the set of features to highlight. High level view requires parsing to link features and files.

VFD is the core of the plugin XToF, it contains SPLAR and the main view. Most functions can be accessed from this view. This plugin does not do the tagging nor the parsing. This role is delegated to the plugin called parsing. However, this plugin is responsible for mapping tags to the features.

This plugin creates a new waypoint in TagSEA: features. In the TagSEA view, it is possible from the tags to access the location of the waypoints in the source code but also to the features in the VFD Viewer. The only data that VFD memorizes are the state of a configuration. The feature model file is saved by using its path and is saved into the project property. The display of the feature diagram is used to access most functions of XToF; they are located in the same Plugin. Others functions like pruning, parsing files, highlighting tags and are delegated to the parsing plugin.

The *parsing* plugin takes care linking TagSEA tags and the features for language independent parts and provides reusable tools for the specific language plugin. It is responsible for highlighting a given range of characters. It also provides tools that can be used by language specific tools like solving interior of tags, generating parsers, pruning, auto-completion. For anything specific to a language (for example defining parsers for files), the plugins delegates it to the plugins Java/C. It is responsible for associating features tagged to the waypoints, the source code and highlighting scope of tags.

The plugins *Java/C* two plugins are responsible for language specific functions. These plugin only contain specific and restrained tasks. They are called by *Parsing*.

- Defining file parser
- Defining tags syntax
- Resolving associations between a waypoint and a portion of source code with the use of the abstract syntax tree.
- Autocompletion
- Link between files and tags

The plugins *Filters Java/C* add functions to create a filtered view of files which contains features selected. They can also annotate the names of the files instead of hiding them. These plugins are also specific to languages because they are adapted to the types of files and elements that depend on the language.

XToF provides two views to help the developer in better understanding how the code has been tagged and how features are scattered through the code. This section will be discussed at length in the guided tour of *section 7*.

6.3.2. TagSEA Documentation

This section will now describe the architecture of TagSEA, as desired by its developers. As previously cited, TagSEA uses the mechanisms of plugins provided by Eclipse. Each plugin can declare an interface that can be called. It can also provide extensions points that other plugins can use to extend TagSEA.

The better way to understand how TagSEA works, is to analyze the dependencies between plugins. A plugin depends on another if it needs to call its interface and methods. A plugin which extends another one is called by the plugin defining the extension (almost everything in Eclipse is a plugin[5], see Eclipse website[6] for documentation on Eclipse plugins). To analyze dependency, a tool was used, PDE Dependency View developed by Ian Bull[7] from CHISEL. The figure created by the tool is put into annex *B*. Understanding how TagSEA works with plugins, helped in designing XToF for Eclipse.

There are two sets of plugins, the core plugins, that are needed for TagSEA to work and extra functions. The extra functions are presented as they enabled us to understand how TagSEA could be extended and how XToF could use the same mechanism. The two next graphics represent the dependency. These figures are available in larger format in Annex C. The first drawing (*figure 6.14*) contains the core function of TagSEA.

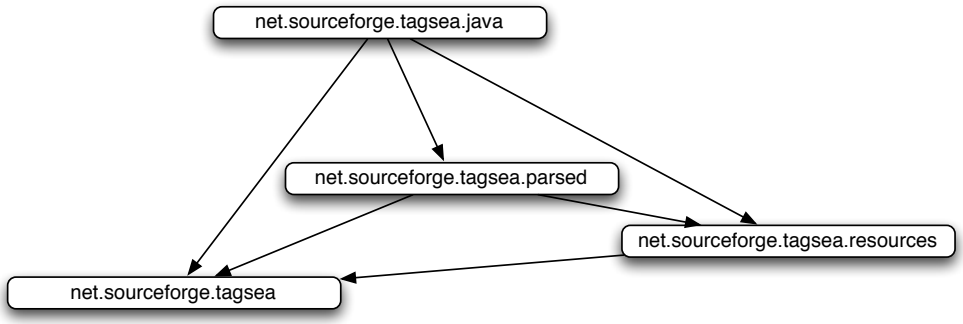


Figure 6.14: Graph of dependencies of TagSEA

- *net.sourceforge.tagsea* is the core of TagSEA, it provides the management of the data model, it links the data to the main views and provides extensions points.
  - It contains the data model. The data model consists of three objects: the tags, the waypoints and their relations. Tags are simple words or hierarchical words. Hierarchical tags are regular tags, but they are interpreted differently and displayed hierarchically. Waypoint is a location where a tag exists and has a location and metadata. Main metadata are the author, the date, the message and attributes. Attributes are fixed by the plugin using the waypoint extension. Concerning the relations, a tag registers every waypoints where it appears and a waypoint registers every tag that it contains. Therefore the developer can navigate from tags to waypoints and vice-versa. TagSEA use events to notify others plugins of modification to the data model. Any plugin defining a new kind of tags is aware of the modification and can adapt the tagging in its function. TagSEA core does not record Waypoints and Tags. They are added by the plug-in providing waypoints during loading. ParsedWaypoint plug in records in a XML file to avoid loosing time. If Eclipse crashed, it would re-parse the source code.
  - Links the data model to the views. The views are the lists of tags and the lists of waypoints. It also propagate modifications using the event mechanism.

---

[5] http://en.wikipedia.org/wiki/Eclipse_(software)

[6] http://www.eclipse.org/documentation/

[7] http://blog.ianbull.com/2007/08/pde-dependency-view-soc.html

- This plugin also declares some extension points: waypoints, filters and starts. Filter provide a way of displaying only waypoints answering a criteria. The start is a method to allow others plugins to start before TagSEA. Finally waypoint is the extension point to declare a new kind of waypoints. The plugin filling an extension point of type waypoint, must define some parameters regarding the waypoint, such as the waypoint is modifiable. The plugin can thus be called to realize the operations. For example if a tag is modified in TagSEA, the plugin is responsible for providing a propagation mechanism.

- The plugin *net.sourceforge.tagsea.resources* defines a type of waypoint for any resource located in the workspace, as defined in the previous paragraph. This waypoint has two attributes: the beginning of the waypoint and its end. It is the waypoint as used in tagging.
- The plugin *net.sourceforge.tagsea.parsed* extends the previous by defining other attributes. It contains all elements of waypoints that are parsed. A parsed waypoint is a waypoint located in a text file, in the source code. It realizes parsing at project level and provides tools to be re-used by actual parsers.
- The plugin *net.sourceforge.tagsea.Java* realizes the parsing for the Java language, separates comments and analyzes them. It uses resources given by the parsed plugin. It is also responsible for the auto completion in Java.

The others plugins are extra functions that are not installed by default. The graph in *figure 6.15* describes their dependencies.

- The plugin *net.sourceforge.tagsea.tasks* provides integration to Eclipse Tasks
- The plugin *net.sourceforge.tagsea.url* defines a new kind of waypoint for URL.
- The plugin *net.sourceforge.tagsea.breakpoint* defines breakpoints when debugging a new kind of waypoints.
- The plugin *net.sourceforge.tagsea.c* is similar to net.sourceforge.tagsea.Java but for C code source.

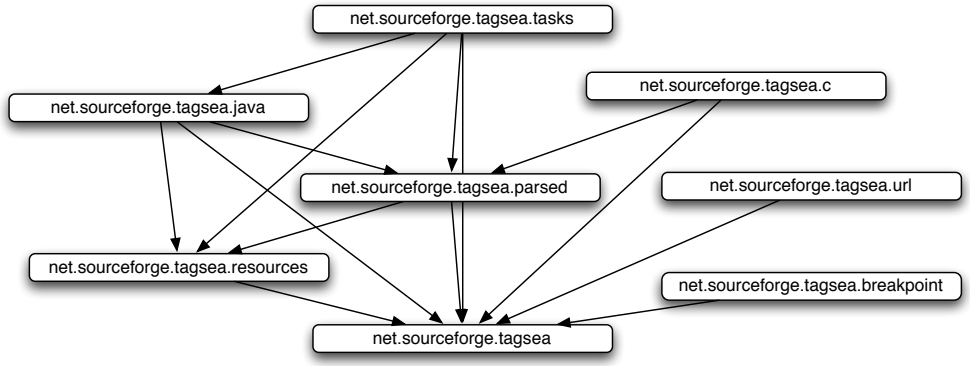


Figure 6.15 Graph of dependencies of TagSEA with extras (see annex B for full size)

TagSEA also provides extension points to allow new plugins to extend its functions.

- The first extension point is called *net.sourceforge.tagsea.waypoint.* It has already been discussed before, it enables new kind of waypoint. It provides information on the information to display to the user, and to which classes must be used for the user interface and for the Waypoint.
- The second extension point is called *net.sourceforge.tagsea.parsed.parsedWaypoint* and provides waypoint for parsed files. Parser provides default implementation for immutable waypoints. It uses a Waypoint descriptor to create the waypoint.

6.3.3.Description of Plug-ins Interactions

This section describes how the plugin interacts betweenTagSEA When a tagging is added in the source code, TagSEA analyses the files

The advantage of presenting the pruning is that, the same plugins are used in the pruning as when the scope is highlighted.The next graph describes the roles in the case of

associating the portion of source code to a tag and the source code. The visitor pattern must be used to go through the abstract syntax tree.

To highlight the portion of source code associated, the user must select it from the feature diagram (plugin *VFD*). Then VFD asks for every tagging containing the specific tags. Using the list, the *Parsing* plugin interrogates the language specific plugin based on the file extension. The *Java/C* plugin interrogates the Eclipse Framework to get the AST of the source code and using the imposed visitor pattern, it searches for the associated node (see *figure 6.16*). If found, it is sent back and is used by the *Parsing* plugin to highlight the associated source code.

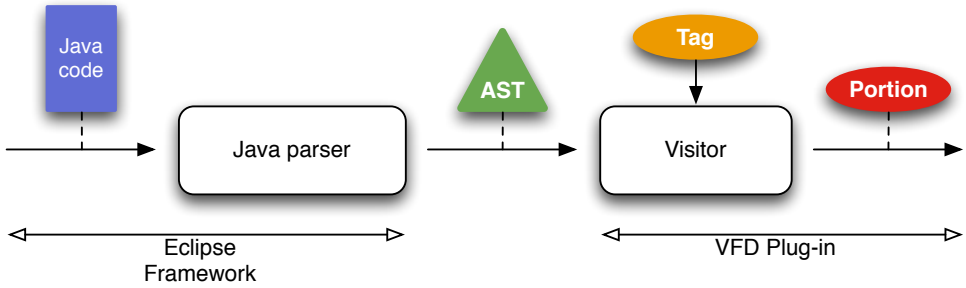


Figure 6.16: Plugin interaction for pruning

When pruning a project, XToF asks TagSEA for all taggings present. For each of them, it uses the associated tag to determine if the associated source code must be erased. If yes, the node is erased from the AST. The AST is provided by the Eclipse Framework that also parses the Java source code.

6.3.4. Data Persistence

TagSEA saves tagging and tags. XToF does not save the tagging. It only keeps in cache the scope of the tagging. The feature model is partially under an XML format and is handled by SPLAR. XToF also caches data to accelerate treatment like opening children of a node. However no informations is saved persistently by XToF. The file is imported into the folder and its name and path are registered into the project properties of Eclipse. Therefore when closing this information is kept.

When configuring a feature diagram, the configuration can be saved in the current state. These states are saved to be restored later when asked by the developer. This information is stored, by using the Eclipse framework, in the hidden folder reserved for the plugin in the workspace. This information is private to the developer.

6.4. Extendibility to New Languages and Feature Model

XToF provides a mechanism to use tags in other *languages*. It requires the creation of a plugin that uses extensions points defined by TagSEA and XToF. XToF provides preconfigured classes to facilitate the adaptation for others languages. XToF allows the developer to use a different plugin for Java and C than the two provided, as a configuration panel lets the user choose which plugin to use to realize the tagging and the pruning for each file extension.

XToF uses Java interfaces to get access to feature models. Therefore, it is possible to use another feature model system. However, this requires modifications to the plugin in itself. The reason for this is that the way Eclipse framework works, the display needs to be adapted if a new feature model is displayed. This tasks would however only take a short amount of time.

6.5. Summary

This chapter first quickly described what is XToF. Then it explained the different decisions made before implementing the tool. They were about the backends used to facilitate the implementation, the scope of tags, the names of features used in the tagging. Then the architecture of the tool was described. It also provided a description of TagSEA architecture.

7. A Guided Tour through XToF

This chapter is a guided tour through XToF. The functions presented here are implementation of the requirements found in *section 5.3*. The processes covered by XToF are the tagging, program understanding, configuring, and pruning. They have been described in *section 5.4.2*. For each of the functions, the mechanisms implemented are described.

When developing the tool and its support mechanisms, the decision was taken to re-use most of the existing mechanism of Eclipse. The goal was to reduce the time needed to develop the tool and to reduce the work needed by the developer to learn how to use the tool by using mechanisms he might already be using in his IDE.

7.1. Tagging Support

To achieve its support goal, XToF has different functions available. Some of the functions described here have already been presented but are reproduced here as their presence makes sense in the guided tour.

7.1.1. Feature Diagram Display

XToF displays a feature diagram as a way to help the developer remember the software product line. It is not displayed as graph, but as a hierarchical list like those already in use in mosts IDE (see *figure 7.1*). First, by using a similar feature diagram, the approach offers a consistent information through all the processes of software product line information. Instead of having to remember the name of each feature, the developer can concentrate only on the group in which the feature should be, which would be facilitated by adequately naming the features, and avoid having to read a complete list. A feature diagram can be loaded to associate it with a project (*R1* and *R2*).

As the developer never works on all features at all time, the choice of opening and closing elements of the hierarchy helps him to focus on a few features and not be distracted by a long list of features irrelevant to his current tasks. The other advantage of using a hierarchy instead of a graph, is that it takes less room. Therefore the developer can keep it open while developing and thus have constant access to all XToF functions.

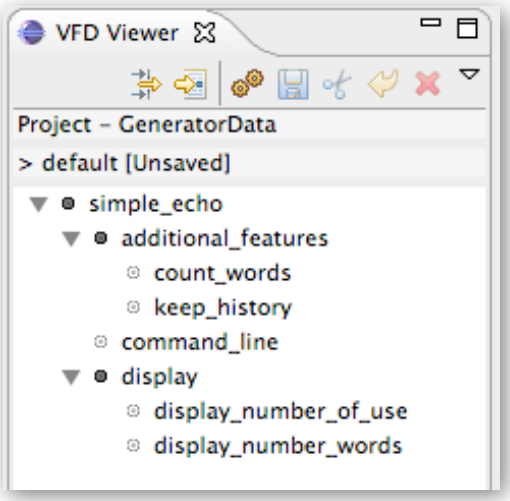


Figure 7.1: VFD Viewer: display of the feature diagram

7.1.2. Feature Name Checking

When typing the name of a feature, it is easy to misspell it. Therefore, the tagging won't be correct, and the feature won't be associated with the portion of source code. It would result in an incorrect pruning. Each tag name is checked and compared to the list of features (*R4*). If a tag name doesn't represent a feature name, or the feature name

is incorrectly written (by example, it contains a space) then an error is displayed. An evolution of the approach would be to provide a list of errors in feature names such as already exists in Eclipse for standards errors.

7.1.3. Auto-Completion

When tagging a portion of source code, it may be hard to remember the complete name of a feature. To help the developer in remembering and reducing the causes of errors, XToF suggests the name of the features (*R5*). When the developer calls the function, a list of every feature is displayed by their short name and can be added to the current tag by selecting. If the developer has already typed some characters or the beginning of a long name, XToF proposes a list of features, or groups, that correspond.

7.1.4. Scope Highlighting

For any tag, the developer must be aware of how it is associated with source code as it will have an influence on the pruning. The tool can display the source code associated with a given tag (see *figure 7.3*). Eclipse provides mechanism to see where a variable is used: it highlights each occurrence of a selected variable. XToF uses the highlighting to display the associated source code (*R7*). As Eclipse does with a variable, XToF enables the developer to select a feature in the feature diagram and each portion of source code associated will be highlighted in the open editor. Eclipse provides some options to enable the user to adapt the highlight (see *figure 7.2*). *Section 7.6* will compare the scope highlighting and other visualization mechanisms with the view presented in *section 4.1*.

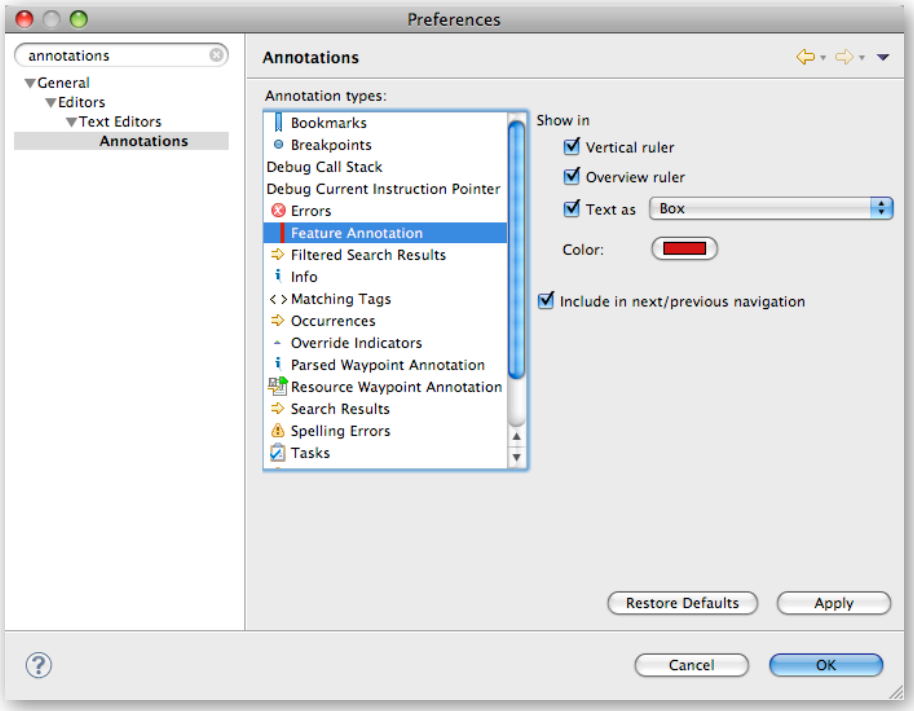


Figure 7.2: Options to change the highlight

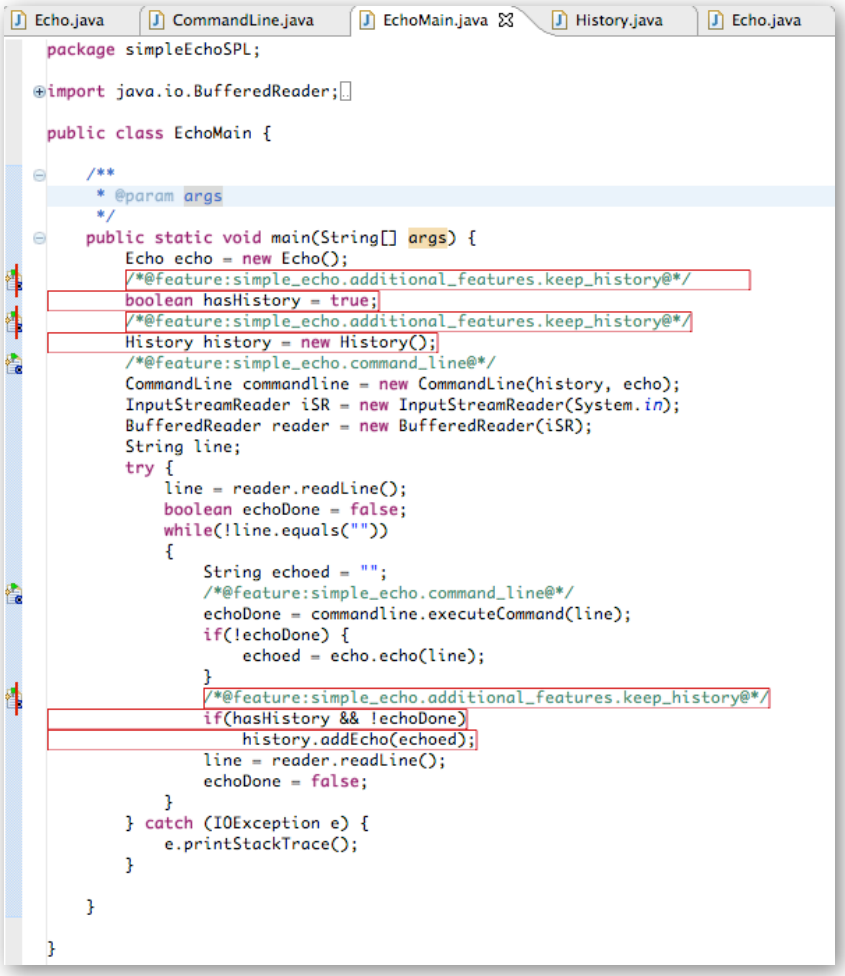


Figure 7.3: Scope highlighting

7.1.5. Associated Features Display

The previous function enables the developer to see which portion of source code is associated with a feature (*R7-R8*). Sometimes as, it may be difficult to understand how the association is made, this can be done in reverse. By putting the cursor in the source code, XToF highlights the features which are associated with the source code at that position and then the scope highlighting is done for these features. The *scope highlighting* (*figure 7.4*) and the *associated features* (*figure 7.5*) enable the developer to check for the tagging of source code in both directions.

Figure 7.4: Associated feature given the cursor

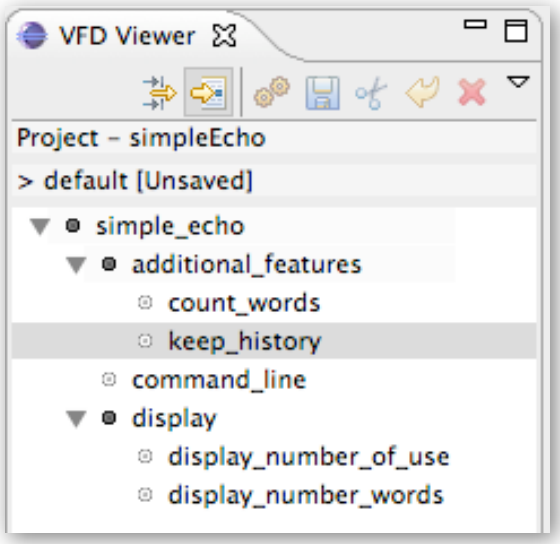


Figure 7.5: Highlighting the feature(s) associated

7.2. Configuration Support

During the configuration process (see *figure 7.6*), the developer must select or unselect features. A valid configuration is a set of selected features and a set of unselected features where each feature is either selected or unselected and all the conditions in the feature model are respected. This section covers requirement *R10*.

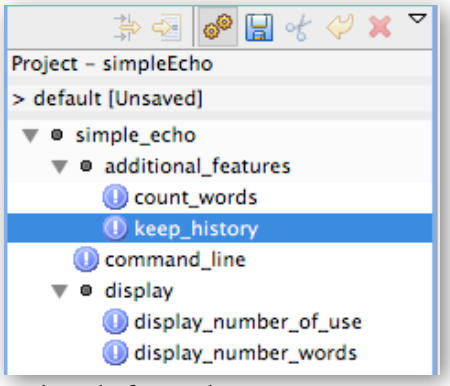


Figure 7.6: Configuration mode in the feature diagram

7.2.1. Feature Diagram Configuration

XToF uses the same feature diagram displayed as a quick and unified point of access to both program understanding methods and configuration mechanisms. Therefore, the developer has no difficulty in finding the name of the feature he wants to select or unselect. While configuring, a bar indicates the percentage of features already selected/ unselected.

7.2.2. Select/Unselect a Feature

Once in the configuration mode, the developer can access some buttons to select or unselect the features (see *figure 7.7*). If a feature is already selected (or unselected) it is displayed. If the choice has already been done, the developer can toggle the choice (see *figure 7.8*). For example, if a feature was selected it can now be unselected. SPLAR then makes the corrections to other features to keep a valid configuration.

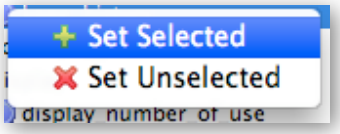


Figure 7.7: Selecting or unselecting a feature

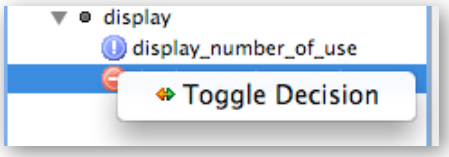


Figure 7.8: Toggling a choice

7.2.3.Propagation of Selection

At each step of the configuration, XToF propagates the choice using SPLAR. SPLAR computes the required (un)selection given the new choice, i.e. if a feature is selected in an *xor* group, then as one and only one feature can be selected, its siblings must be unselected. Then XToF display the features in the new state. This mechanism forces only valid configurations and reduces the number of steps needed to realize a valid configuration.

7.2.4.Minimal Set of Features

As Boucher *et al.* [3] stated, one possible source of error is tagging the declaration variable, type, class or function with a feature and using it with another feature or no feature tagged. When pruned, if the feature containing the declaration is not selected, the declaration is removed from the source code but some instructions may still use it as they have not been correctly tagged and pruned. This results in errors. To prevent such issues, Boucher *et al.* [3] have developed a design rule: «*Each feature can only use variables, functions and types declared by itself or by its principal dependencies.* » where principal dependencies are the features that must always be present with a given feature (also called *the minimal set of features associated with a feature*). The developer then has only to generate the pruned project with the minimal set of features for each feature of the feature diagram. The developer can right click on a feature, in configuration mode, to prune the project using only the minimal set of features associated with the right-clicked feature (*R13*).

XToF can compute the minimal set of features that are always present with a selected features using the heuristics given in the same article. As this is a heuristic, it is not proven that it will always select every feature and may miss some features that may always be present with the selected features. Therefore, XToF also implements this research by using the feature model in SPLAR. SPLAR can provide every feature that will be always present with the selected features. The disadvantage of SPLAR is that it requires multiple computations that in the case of large feature models may require some time. XToF provides the developer with the choice of using one or the other (see *figure 7.9*).

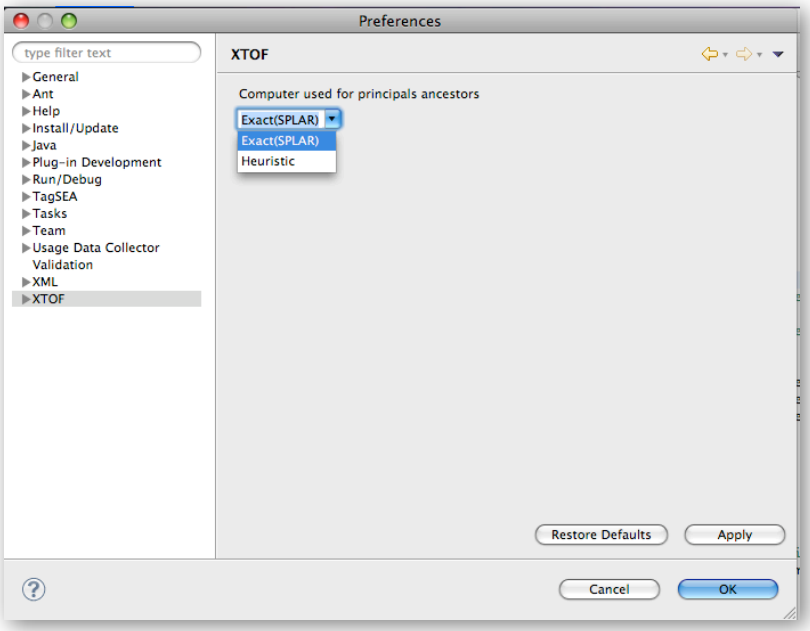


Figure 7.9: Choosing the computing mechanism

7.2.5. Save the State of a Configuration

Early testers of XToF, proposed implementing a mechanism to save the current state of a configuration (see *figure 7.10*) in order to avoid having to reconfigure the same set of features several times when doing series of tests. Therefore, the state in which the configuration is done can be saved and recovered later (*R11*). This is kept in the workspace, so that the next time the IDE is restarted, the states are still saved. On top of this mechanism, classical functions like *undo last step* and *reset* configuration are implemented.

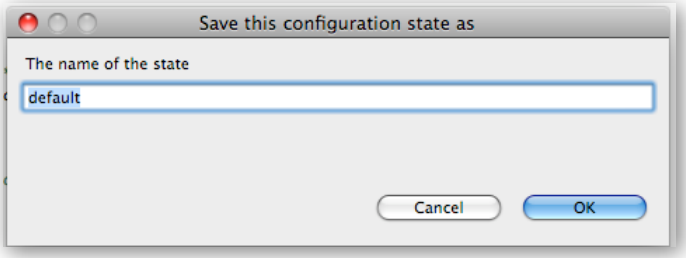


Figure 7.10: Saving the state in the configuration mode

7.3. Pruning Support

Pruning is integrated into XToF and avoids the necessity to use an external module. This section covers the requirement *R12.*

7.3.1. Prune into a New Project

When a configuration is complete, XToF enables the pruning. The developer must then choose a name for the pruned project. Therefore, the whole pruning approach is integrated into the IDE.

7.3.2. Pruner / Scope Resolver Selection

XToF contains a pruner and a system to resolve scope. However, sometimes, due to some constraints, the developer may have to use a different pruner, for instance an external pruner. XToF provides options to choose another plugin to prune (see *figure 7.11*). The same approach can be used with the scope resolution, i.e., the association between a tag and the source code.

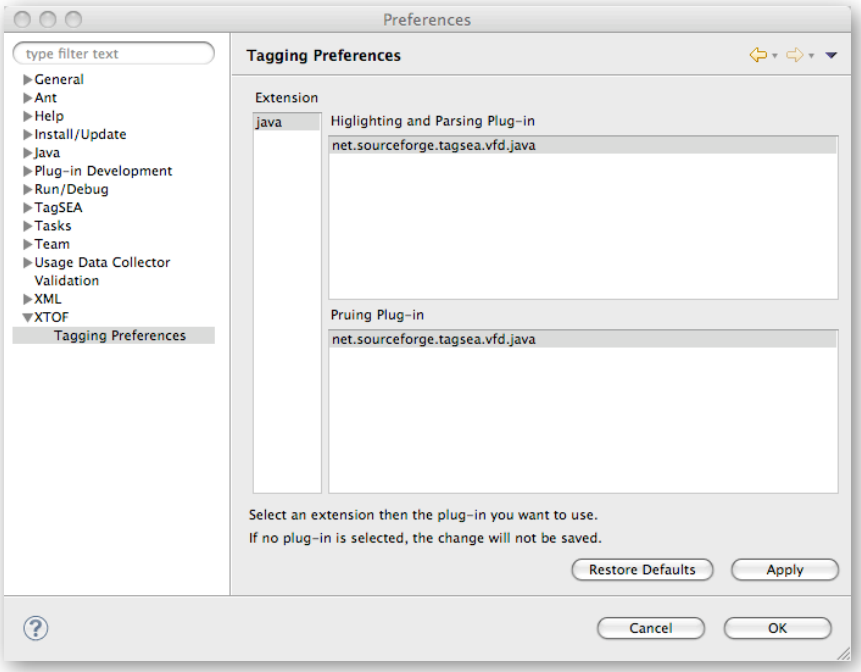


Figure 7.11: Options to choose which plug-ins to use.

7.3.3. Adaptable to New Languages

As XToF already provides two languages, C and Java, other developers may want to use the tagging approach in other languages. Therefore, every operation that is specific to a language is repeated in a specific plugin. Developers can create plugins for new languages. In the case of the C language, it only took a few days to develop a plugin, include the time to understand the IDE frameworks for C.

7.3.4. Information about Pruning

Using the recommendation of the IDE, a windows displays the percentage of files already pruned. It also allows the developer to cancel pruning before it is finished.

7.4. Program Understanding

Tagging is limited to one file at a time. Program understanding (also called program comprehension) is a task aimed at helping the developer to understand a software. In this context, the program understanding is limited to understanding of how a program is featured, i.e. how a programmed is tagged with features. As Storey [38] describes, the program is understood by building a model at different levels of abstraction. XToF provides three levels, one inside a file, another at files-folders levels and the last, aiming at a project level, which will be seen in *Chapter 8*.

7.4.1. Source Code Level

The file level is done by the scope highlighting and the associated features. It provides a way to understand how a file is tagged with features by displaying the source code associated with features and vice-versa.

7.4.2. Files and Folders Level

The files and folders level is aimed at enabling the developer to understand which features files are tagged with and vice-versa. Three mechanisms are implemented.

File name labeling

By selecting a set of features in the feature diagram, every files that contains at least one of these features in a tag is labeled. In the file explorer, a tag is followed by the name of the file indicating that the file is tagged. This enables the user to know where a feature is present (see *figure 7.12*).

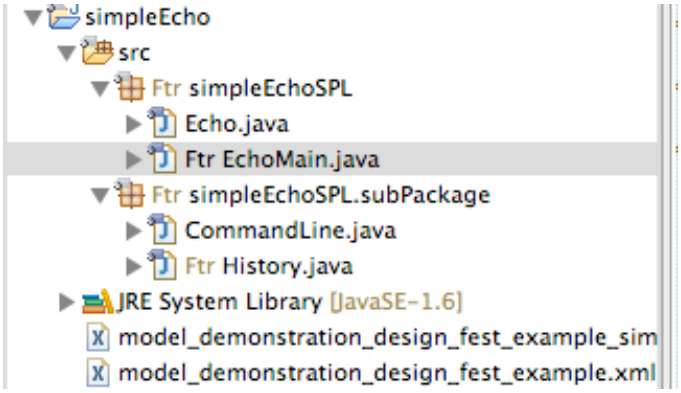


Figure 7.12: Labeling of file name

Hide files

This function uses the same principle of feature selection but instead of labeling, it hides files that do not contain any tag with a selected feature (see *figure7.13*). Thus, it enables the user to focus only on files implementing the set of features. It reflects the views of Kästner *et al.* [21] showing only artifacts related to the features, the realization view and the context view (see chapter 4 for visualizations). The realization view focuses on only one feature while the context can deal with more than one feature by coloring them. The hide file mechanism doesn't use the color metaphor.

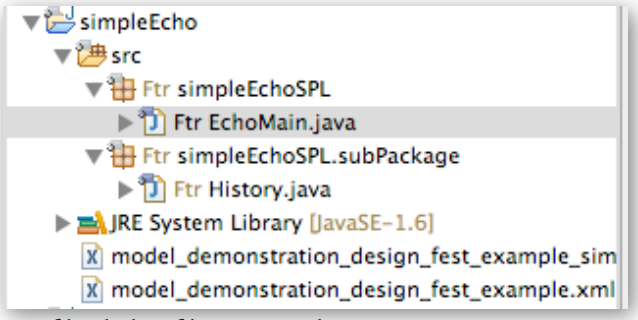


Figure 7.13: Filter on files, hiding files not tagged

File selection

The file selection provides the opposite mechanism. By selecting a few files (inside the same project) in the file explorer, XToF can highlight the set of features that are tagged in at least one of the selected files. It is founded on the same principle as the associated feature function.

TagSEA

As XToF works as a plug-in of TagSEA, it can use its function to improve the support to the user. This includes displaying all features as tags, and for selected tags, the list of every file and location of the tagging (see *figure 7.14*) (*R8* and *R9*). Therefore, it is another way to understand how multiple files are tagged. Selecting one of the taggings or the features can then start the scope highlighting in XToF. The developer can, by selecting the features from TagSEA, display a list of all tagging locations and highlight them.

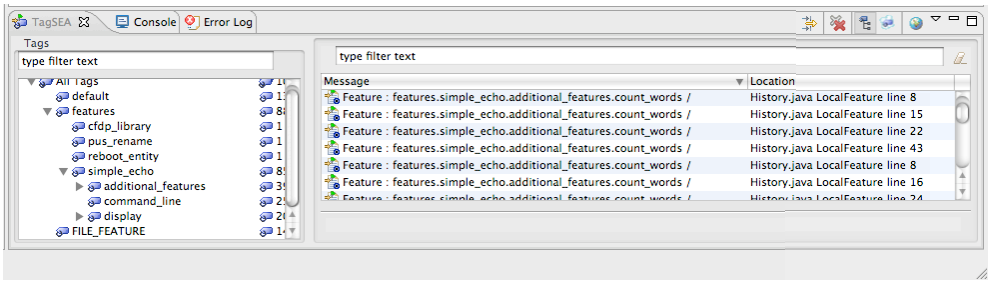


Figure 7.14: TagSEA in conjunction with XToF

7.5. Project Level Visualizations

Heidenreich et al. [13] presented four views as visualization techniques which were described in *section 4.1*. This section will explain how XToF uses their ideas. The fourth, the property-changes view is not presented as it is out of scope. This section covers the requirement *R14*.

- The *Realization View* displays only the source code associated with a feature. XToF provides the scope highlighting to underline source code that is associated with a feature. As it highlights the associated source code instead of hiding the rest, it enables the developer to understand how this code relates to the rest. The *labelling* and the *filters* extends this function to more than one file.
- The *Variant View* displays only the source code which will be part of a specified product. To achieve this task in XToF, the developer must make the configuration and prune the project. The pruned project will provide the same information as the variant view. The future version proposes to use the folding mechanism of Eclipse that hides parts of the code. The developer would have the same information without having to prune.
- The *Context View* is the realization view extended to more than one feature. XToF provides the same function for one or several features. The scope highlighting, labelling and filters can be used for one or more features. However, XToF does not use colours as some peoples can't see them correctly and if features overlap then mixing colours is problematic [13]. A portion of source code is highlighted if any of its as-

sociated features is selected for scope highlighting. The same idea is used for the others mechanisms. Future versions of XToF should provide the choice for the developer to highlight only portion of source code associated with every selected feature and similarly for each mechanism.

As a way to support the developer in his task of understanding feature implementation at a project level and feature interaction, a project level visualization was developed. Its purpose is to help the developer in understanding how features are implemented given a set of feature which files are tagged with these features and vice versa. It also aims at displaying the interaction between features, i.e. which features are tagged in the same files. It helps the developer identify these files and allows him to check how exactly the features interact inside the file.

Two project level visualizations were developed. The first is based on a graph linking features and files implementing them. The second, uses a matrix to represent features and files. They are presented in the two next sections.

7.5.1.Zest View

The first visualization that was designed was called Zest view, from the name of the library used to draw it. It was based on the use of a diagram showing the relations between features and files. The advantage of using graphs is that many developers are familiar with them. They are effective in showing relations.

There are two kind of nodes, the feature node and the file node. A file node is a node corresponding to a file; each file has one and only one node; and a feature node represents one and only one node. A feature can only be linked to a file and vice versa. A file is linked to a feature if the file is tagged with the feature. The graph enables developer to see features interacting through the file nodes that are linked to both features.

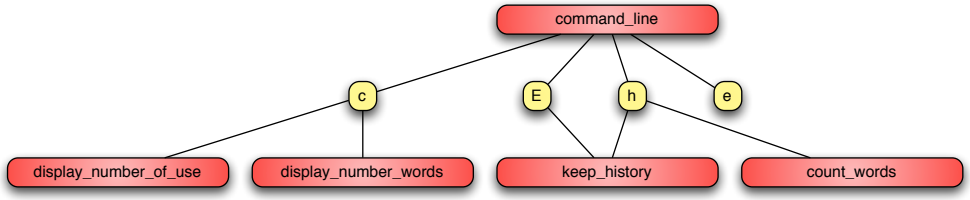


Figure 7.15: Mockup of Zest View

The example displayed uses the data from an example (cf. chapter 8). To provide a clearer diagram, the file nodes are coloured in a different colour and their name appears only when the mouse hovers them (see *figure 7.15*). Zest provides different layout to arrange the nodes (the mockup display does not use Zest). The goal was to obtain features nodes scattered throughout the graph with file nodes gravitating around the features nodes to which they are linked thus making for a clearer picture. However, due to the complexity of arranging edges, no satisfactory result was obtained.

The issue comes from the fact that Zest uses directed edges and was not conceived for multiple roots (features nodes). Several attempts to choose a unique root and organize the edges based on this while keeping their logical meaning were made but didn't achieve a satisfactory result. On top of that layout question, other issues (see *Evaluation*) pertaining to this visualization rendered it inadequate for the goal and another visualization was developed.

The developer can consult the graph to search for interaction between features by searching for file nodes that are common to more than one feature node. For example, in *figure 7.15*, the feature *display_number_of_use* interacts with the feature *display_number_words* through the file *c*.

For a given feature node, the number of links indicates how much the feature is scattered. This can help the developer locate a high complexity. The feature *command_line* is linked to four file nodes (each file in this example), which could be an indication that this feature implementation may be too complex.

Some disadvantages of this visualization is that the information given is binary (yes/no the feature is implemented in this file), large graphs are difficult to read and do not provide any mechanism to search for specific elements.

Given theses disadvantages and the layout issue, this visualization was not included in the final version. Instead, another visualization was first drafted, then developed and included in the final version, the *concern view*.

7.5.2. Concern View

Concern View originates from a visualization called *ConcernLines* by Treude *et al.* [42]. Its purpose is to help the developer understand co-occurring concerns, *«the evolution of a software system requires understanding how information about the release history, nonfunctional requirements and project milestones relates to functional requirements on the software components.»*. It displays each concern on a timeline and uses colour intensity to indicate the relevance of the concern. ConcernLines takes a CSV file in input, i.e. a matrix of numbers, which enabled a fast mockup of the Concern View for XToF.

Concern View displays a table where each row is a feature and each column is a file (see *figure 7.16*). A given cell indicates the number of times the feature (the line) is tagged in the file (the row). The colour intensity varies from the lightest (the smallest number of tagging) and the darkest (the biggest number of tagging) (see *figure 7.17*). The colour scheme is mapped according to the minimum and maximum of the matrix then displayed. To complete the information given by the colour, the number of tagging is displayed in each cell. If this number is null, then the colour is white. *Figure 7.16* is the initial concern view when a project is selected. It shows the main folder (here *src/*). As feature is selected, it is in blue (*count_words*). *Figure 7.17* is the concern view for the *src/simpleEchoSPL* folder. The blue cell is due to selection. Its information is displayed under the view.

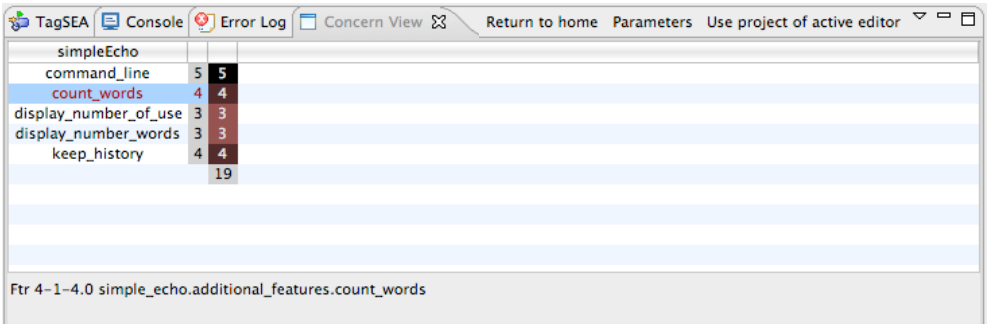


Figure 7.16: Concern View - Home of a project

Figure 7.17: Concern View - Inside the folder src

For each feature, a special cell is added. It is not linked to a specific file or folder, but to the project in itself. It indicates how many time this feature has been tagged in all files. The same idea is repeated with the files or folders. These two sets of special cells provide informations independently of the row or the column and do not influence the colours.

When clicking on a cell, line or row, different information is displayed, like the long name of the feature, the name of the file and its path if it can be applied, the number of taggings, the number of files tagged and the average tagging per file. This information can help the developer understand how a feature is scattered through out the software and through out a file. For a resource, only the name of the resource, the number of taggings, features and average tagging per features are displayed. For a cell which is at the intersection of a feature and a file, the display adds the number of taggings of the feature in the file and the name of the feature (see *figure 7.18*). For a feature, it displays, its name, the number of taggings and files tagged and the average tagging per file (see *figure 7.19*).

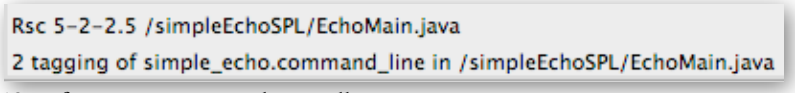


Figure 7.18: Information associated to a cell.

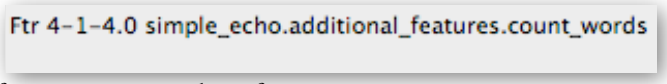


Figure 7.19: Information associated to a feature.

The *Concern View* only display the files and folders of the current folder. It helps reduce the number of elements displayed at any time. If there are sub-folders, then the sub-folders are displayed instead of theirs files and the numbers correspond to the agglomeration of files numbers. A navigating mechanism is available to switch to a sub-folder and vice-versa. In *figure 7.20*, the *subPackage* folder is displayed. The feature and file cell always display the same values to enable comparing a cell to all taggings of the feature and file.

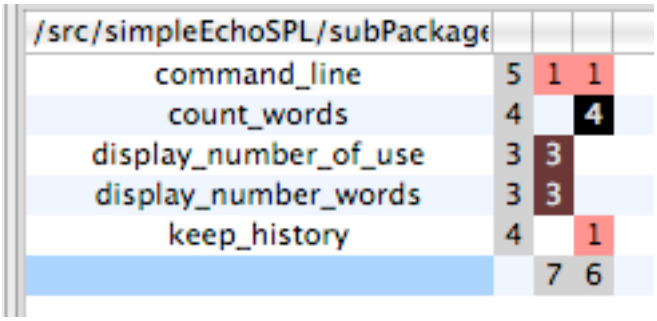


Figure 7.20: Concern View in the subPackage folder.

Concern View also provides other mechanisms to help the developer in finding the desired information. First a limit, enable the developer to limit the number of features or files displayed. Second, columns and rows can be sorted. With these two mechanisms, the user can search for a specific information like the ten features that are the most tagged. The filters and orderers are described later.

How to Interpret the Concern View?

Concern View is a matrix, it can be read in different ways, by reading information from a specific cell, from a column, from a row or by colour. Each provides different information to the developer. The information displayed by a cell was described before and won't be repeated. The visualization can help the developer understand how the features are implemented at a project abstraction level but can also be specifically used to search for potentially complex feature implementation.

When reading the Concern View one row at a time, the developer focuses on a feature. He can understand how the feature is implemented from a project level abstraction. For each feature, the files that contains the feature are coloured. The view also displays how many times it is tagged in each file. It is then possible to see how scattered the feature is.

The last row which contains the special cell containing the number of taggings for each file or folder can be used to pinpoint files that have a potentially complex feature implementation or feature interaction. Then the developer can check the reason. It may be due to only one feature, in which case this feature is tagged a number of times in the files and may require a review of its implementation in this file and factorization, or if it may be due to a set of features.

By column, the developer focuses on a specific file or folder, so he can understand feature interlacing. It displays which features are tagged in a given file. Not only can he see which but also the number of taggings, which is a supplementary information compared to the Zest View.

By searching for columns that contain more than one coloured cell, i.e. not white, the developer can understand how features are interlacing. However this research may take long if it is done manually, particularly in the case of very large software.

A special column contains the number of tagging for each feature. It can be used to search for complex feature implementation. The developer can then check the line for the reason and pinpoint the localization of eventual improvements, that may be necessary.

As the colour of a cell indicates the relative number of taggings compared to other feature and file couples, the developer can search for particular values. By searching the set of dark cells, he can find the couples of files and features that have a larger number of taggings than other couples. Within the set of light cells, the couples with a low number of taggings can be found. These are the extremities of tagging numbers.

An other use of the colour is to make a relative comparison. If the developer searches line by line or row by row and only the numbers are displayed, he may miss the fact that the number displayed is comparatively normal and not the largest. The colour helps to visually compare features and files taggings.

One advantage of using a table to display the information is that it can fit larger cases than graphs. As the scope of this work limits software to a reasonable size, it provides a readable visualization. However achieving a specific task, like finding the features which most scattered, could only result from a mechanism to reduce the number of features and files displayed to a smaller set. Concern View then provides two mechanisms to help the developer in displaying only the information he want; these are the *limit* and *sort* mechanisms (see *figure 7.21*).

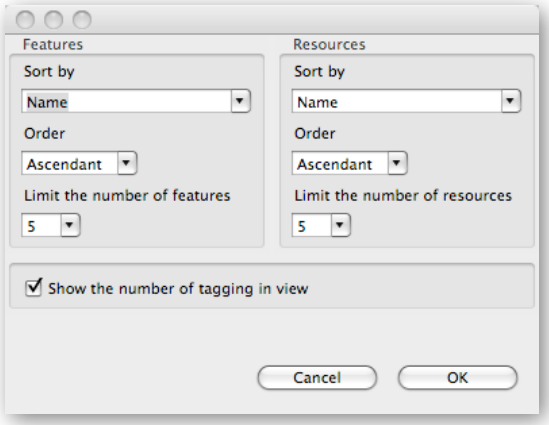


Figure 7.21: options of ConcernView

Option window for sort and limit.

Limits enables the user to display only a limited number of features or files, for example only ten files. While the sort provides different algorithm to order features or files in ascending or descending order. Together, they can be used to display only the ten most scattered features for example. These mechanisms have an impact on the number of and order in which the features and the files are displayed, but not the number of tagging and colour. Next the different sort options and their purpose are explained.

- The *name* sort provides a simple (anti-)alphabetic sort, it can be used from a global perspective or to search for a specific file or feature.
- The *number of tagging* option sorts according to the number of times the feature has been tagged or the number of taggings in the file. This can be used to pinpoint features that have a complex implementation, or files that have a complex feature implementation. It doesn't take into account if it is in different file or by different features.
- The *number of element* sorts the feature by the number of files in which they are tagged. This is very useful to search for scattered features. In the case of the sorting applied to the files, it sorts by the number of different features tagged in it. This provides a range of complex feature interactions in each files. Similarly to the number of taggings, it doesn't take into account the number of taggings by feature or file, i.e. several features may be tagged only once in a file or several times.
- The *average number of tagging* sort option orders the feature by the average number of tagging per file and the files by the average number of taggings per feature. This can be used to locate complex feature implementations or interactions by taking into consideration both the number of element and the number of taggings sort options. It can avoid intermingling cases where the scattering is due to an obligation to tag the feature in several files, since it marginally touches several aspects of the software, for example in a case where several files are tagged with several features.

The *Zest view* was not included in the final version because of some of its disadvantages. The *Concern View* replaced it and was more suitable for the following reasons.

- Concern View not only immediately states if a feature is tagged in a file without the need to follow several edges, but it also provides a numerical information directly instead of through binary. Zest View could however use the same colour techniques but would loose in visibility because of the two kinds of nodes.
- In Zest View, the colour is only used to distinguish feature nodes from files nodes and can be replaced by the form of the node, in Concern View, it displays supplementary information: the relative range of the number of taggings. However using colour has some limitations. First the number of colours that the human eye can distinguish is limited. However in this visualization, it doesn't have much impact as two close colours that may be interpreted as the same would not result in a major error in the user mind as by definition, the two cells are close. The second is that colour may not be seen correctly by all users. In Concern View, it is not the colour that matters, but its intensity, therefore, not seeing colours correctly or at all has a reduced risk of affecting the visualization. On top of that, the colour is only there to support the number that are displayed and serves as a means to make comparison easier.
- For the same display contents, the *Concern View* is more easily readable. *Zest View* adds complexity with edges. For both views, the sort and navigation mechanisms can reduce the number of elements displayed at any time, rendering it more easy to read. Using a hierarchical file explorer provides a visualization closer to the project instead of nodes that are randomly positioned. It helps the user more easily in understanding the software tagging.
- Find complex feature implementation and interaction: Concern View helps achieve some precise tasks through the possibility of computing requests like «display the ten most scattered features». This can be useful when trying to optimize feature implementation. The *Zest View* could also implement such a mechanism, but *Concern View* is more appropriate as it already displays the value used to sort and limit.
- *Concern View* in the final version of XToF provides only a small number of metrics to display and sort, however it could be easily adapted to use new metrics like the percentage of tagged source code in a file. With a more complex implementation, it could use the percentage of code common to more than one feature. In a different perspective, the Concern View could instead of comparing feature to files, compare features together and compute the number of taggings, percentage of code source, etc., that are common to the two features. These possible evolutions would be particularly suited to the task of understanding feature interaction at a project level of abstraction. *Zest View* does not display values, it could do so by clicking on one node, so understanding the sorted graph may be more complex as the values used to sort are hidden.

For these reasons, the *Concern View* is better at a project level visualization, to an understanding of how features are implemented.

7.6. Summary

When comparing the annotative approaches with the compositional approaches, some disadvantages had to be noted for the former. These were the traceability and the safety. While the second is widely covered by the approach, the first is the one that fully requires the tool support.

On top of helping the developer find the portion of source code associated with a set of features, XToF also supports the configuration, the pruning and the program understanding for features implementation. However in the last one, it is limited and will be improved in the next chapter.

The different support functions help solve the disadvantages and contribute to the lightness of the approach by reducing the work of the developer.

8. Illustration

This section provides two examples. The first refers to the tagging and pruning approach only. The second concerns the support brought by XToF. Their purpose is to demonstrate how the approach fulfills its goal and how XToF can effectively provide support for the approach.

The tagging approach was initially developed in a project at SPACEBEL. This has been described in a paper by Boucher *et al.* [3]. The tagging approach is a method that was successfully used to implement a project. With the pruning, they were able to remove dead code. Tagging enables a fine-grained feature implementation without rendering the code difficult to read. However, a small amount of supplementary time was needed to tag, due to the fact it was applied to an existing project. This example also showed the need for the support brought by a tool, which XToF aims to fulfil.

8.1. SimpleEcho

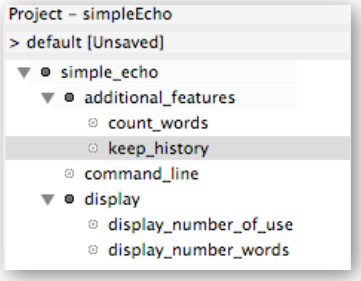
Once XToF implemented enough functions, an example was developed to test and demonstrate its viability. The name of the example project is SimpleEcho. The small software repeats in output what was written in input. It provides also functions like a command line mechanism that can be use to enter command such as displaying some statistics and the history of the outputs.

8.1.1. Feature Model

The feature model has eight features. The root is *simple_echo*. It has three children, one is optional, *command_line*. The two others are OR groups *additional_features* and *display*. First has two children, *count_words* and *keep_history*. *Display* has also two children, *display_number_of_use* and *display_number_words*. There are six constraints in the feature model.

Display_number_of_use and *display_number_words* require *command_line. Display_number_words* requires *count_words. Command_line, display_number_of_use* and *count_words* requires *keep_history.*

Feature model source code is in the annex D.



Feature diagram of SimpleEcho

Children of *display* are features used to display information like the number of times the software has echoed or the total number of words echoed. *Command_line* is the feature enabling the user to enter commands like displaying information and using the history.

The feature model is a small example but complete enough to test functions of XToF. It features children features, mandatory and optional nodes and constraints to test the propagation and the pruning with minimal sets of features. For example the feature

count_words can be unselected and renders the selection of *display_number_of_use* impossible and reduces the number of commands that the user can enter.

8.1.2. Command Syntax

This section briefly describes the syntax to enter commands to provide the reader an explanation of how to use the example. To make the commands available, the feature *command_line* must be selected.

Commands have a general syntax of a two letters word and some of them are followed by a number. There are four commands. Two display information, the others are present to use the history.

SU: *Show number of Use*. Displays the number of times the software has displayed something. It requires feature *display_number_of_use.*

SW: *Show number of Words*. Displays the number of words that have been displayed since the software started. It requires feature *display_number_words.*

RI *i*: *Repeat at Index* i. Repeats what the software outputted the *i-1* time. The index starts at 0. The index *i* must be equal to or bigger than zero. If there is no corresponding display, it warns the user.

RL: *Repeat Last*. Repeats the last output of the software.

8.1.3. Implementation

The implementation is rather simple and is done in four files in two packages. Main package contains the Echo.java and EchoMain.java. The sub-package is called subPackage with CommandLine.java and History.java. Source code of files is given in annex E.

EchoMain.java: Contains the main function and the call to command line, the history and the echo function.

Echo.java: Does a simple echo.

CommandLine.java: Does the command analysis and displays results.

History.java: Records the history and provides access to it.

8.1.4. Supplementary Time Needed

Tagging while coding the echo doesn't add a noticeable supplementary time. As the tagging is done while coding, it only takes the time to use the autocompletion to tag the source code as desired.

The use of the different support mechanisms was useful to provide information on how the software is tagged. It even highlighted a difficulty with the way the cases were tagged and was resolved in XToF. As for the switch-cases statement in languages, the instructions may depend on a break to be runt or not and the abstract syntax tree nodes associated to the instructions in the cases are not children of the case node. This was shown by the highlighted feature.

8.2. Lessons learned

Even if the example was small, it was useful to put in practice the support XToF brings. Here are some lessons learned.

The example showed that using tags to implement software product lines is a light weight approach. It doesn't require a complex software adaptation. It can be done without a large background in software product line and can be accessed by any developer. Having the possibility to never leave the keyboard when coding to tag a feature enables saving of time. Integrating XToF through a feature diagram is a convenient

way to access the features. It integrates well in the IDE and can be used to give support without leaving the current layout.

However, the approach is not a weightless method. It doesn't require the modification of the software and implementation of each feature in separate modules as compositional approaches do, but enables using fine coarse-grained feature implementation. However, locations of source code that are tagged must take into account the fact that the code is erased in some cases and not in others.

For example, it is interesting to see how the simpleEcho example implements the command line option in the instruction flow. First it checks if the input can be interpreted as a command. If yes, then the command is executed. If not, the input is transferred to echo which displays the output. This choice of implementation enabled separation of the implementation of the echo from the command line. That way, the echo function could be replaced by a more interesting feature like a logged echo without having any impact on the command line implementation. Thus if the feature command line is removed, the call to echo is unmodified. If the command line feature is selected, the same call to echo is never run.

In summary this approach avoids the need to adapt the software at a high level, but the source code may be adapted at a coarse-grained level to take the pruning into account.

Integrating XToF through a feature diagram is a convenient way to access the features. It integrates well in the IDE and can be used to give support without leaving the current layout. Concern View showed interesting results and fulfilled its promises. However due to the small size of the example, its full benefits could not be demonstrated.

Tagging with keyboard

Not needing to leave the keyboard to tag a feature is quick when it is done at the same time as developing. However, if the tagging is done on an existing code, a drag and drop approach to tag source code may be quicker. Combined with the scope highlighting it would be closer to the process of retrofitting an existing software to software product line, as it requires that many files be checked, which is usually done with the mouse.

To facilitate the tagging with the drag and drop, the scope highlighting would be used. Using the position of the cursor, XToF could highlight the source code that would be tagged if the button was released. XToF would then place the tagging where it needs to be put, to tag only the highlighted source code. If the developer wants to tag an other portion of source code, he just needs to move the mouse cursor somewhere else.

Minimal Set of Feature

Using the pruning according to a minimal set of features is a useful method of preventing errors in generated product. In the example, it enabled the detection of an error in an import. As the example is coded, Eclipse can automatically add the import of needed classes. The source code that requires the class is tagged as the developer is aware of it, but can miss the import that is automatically added. In some cases it may not cause any error, but in this case the file that was imported was tagged with the file tag. Therefore the file would be deleted in some cases and the import would provoke an error in the resulting pruned project.

In the case of the example, it didn't require too much time to prune for each feature. However if there are more features, an automation could be a lot more quicker. Clicking on a button, the developer can automatically apply the pruning minimal for each feature, XToF could check if the resulting pruned project contains any errors. If any are present, the developer is warned and searches for the source. If there is no error, XToF just displays that the operation is successful.

It could ensure the type safety and call correctness of every possible generated project as stated in chapter three. However, if the original project contains errors, XToF may not be able to distinguish them from the error arising from the pruning.

Size Limit

As the main point of access to the features and the function is a feature diagram, it may become slower to access them if the feature diagram is large. However, Concern View is already capable of handling such sizes through the use of limits and sorts.

Concerning the display of a feature diagram, much research are already done. For example Czarnecki *et al.* [9] propose to use a multi-level configuration for large size feature models. They use a different feature model, part of the global one, for each step of the configuration. XToF could use the same principle to reduce the number of features displayed at any time.

9. Discussion

This work was limited by the time allowed to develop XToF and the time allocated for writing the thesis. Therefore, some points were not included in it. This chapter proposes to explain what could be done as future work. There are two main themes: the methodology and the approach in itself.

This work is not interested in trivial cases, such as feature models where features are all selected or not at all. It also excludes large feature models. As a simple method to circumvent the size, the metric of the hierarchical display of the feature model is used. If the feature model displayed this way becomes too large to be used easily without requiring filtering, searching, etc., then it is not part of the scope.

9.1. Empirical Evaluation

The tool XToF is aimed at supporting the implementation of Software Product Lines. Once developed, it was tested first with developers new to the software product line engineering. Their comments enabled a check of wheter the approach could be light enough to be used by such developers.

Then, the tool was reviewed by developers and researchers in software product line engineering through a tool demonstration paper [12] published at the workshop Vamos'10 (*Variability Modelling of Software-intensive Systems*). While the paper already described most support features of XToF, the published version did not include the project level visualization. It would also be necessary to validate the interest of the Concern View through a user study.

An empirical evaluation should be done to analyze what are the needs of the developer used to SPL and how XToF fulfils them. The user study could also help validate the light weight of the approach by comparing the supplementary time needed to implement an SPL with the tagging approach as opposed to other annotative approaches.

9.2. Missing Features

9.2.1. Independency from Pruning

The tagging approach is, in this work, used in conjunction with the pruning. However, it could be used with other techniques to generate products. The current version of XToF proposes a limited method of using a different module to prune. It should be improved and made completely independent of any reference to a pruning method to enable the export of functions to compositional methods like the export function from Kästner *et al.* [20]. As the author says, it can be used to optimize and fine tune.

9.2.2. Auto Check for Type Safety

One of the issue of the tagging approach is the fact that a typing error can easily arise. They can be also corrected easily but need first to be detected. To help in their detection, the idea of minimal sets of features associated with a feature was implemented from the idea developed by Boucher *et al.* [3]. However, it requires pruning and compilation for each feature of the FM.

As this is a time consuming approach, Kästner *et al.* [19] proposes a formal typechecking of SPL. Their approach intends to detect immediately the possible errors. XToF could reinforce the safety objective by implementing a mechanism that can detect possible errors.

9.2.3. Drag and Drop to Tag

Using only keyboard to tag features while implementing is a quick method. In the case of existing source code, it may be time consuming to have to switch from the mouse to

the keyboard. Instead, a drag and drop method to tag portions of source code should be provided to the developer.

9.2.4. Feature Diagram Creation

XToF does not provide any feature model creation functions. The user has to create a feature model separately and load it. XToF in a future version should propose feature model functions to allow the developer to work only in the IDE. It would make the use of the approach easier by integrating the feature model. The developer would not have to switch to another tool to create the FM.

If XToF provides creation functions, it should also enable their modification. Nonetheless, modifying a feature diagram that has already been used for tagging can decrease coherency between the feature model and the tagging. Therefore, it should also provides a mechanism to propagate modifications to the tagging already made in the source code.

9.2.5. Class View Project Level Visualization

Class View is a continued effort on project level visualization. It uses the class structure as a model for the visualization. The idea is to adapt the existing class diagram to the tagging.

Class diagram with dependencies generation

Existing tools can generate a class diagram from a project source code. They can then display the relations between classes. This information can not only help the developer understand how the software is structured but also the interaction between software classes.

Shrimp[1] is a tool by Storey *et al.* [39] that can be used to visualize software. By parsing the source code of a software, it can generate a set of visualizations. With such a tool the developer can understand how software is implemented.

This idea has already been discussed in *section 4.1.* Heidenreich *et al.* [13] proposed to apply the views to the model themselves. Therefore, when selecting a feature only some elements of the models would be displayed, depending on the kind of view.

The advantages of a such visualization would be that it uses a class diagram as a base model. It helps the developer in understanding more easily how the software is tagged as it is closer to a model he already knows. By hiding classes and dependencies that are not associated with a set of features, the visualization enables the user to better understand software tagging. It helps him focus on some features or classes only. The disadvantages are that it is separate from the source code. It should provide a zooming mechanism as seen in *section 4.3* to link classes to their source code.

9.3. Tool Limitations

The tool provided to support the approach has some limitations.

- The FM associated to a project, can be changed. A new version can be associated. However, XToF only changes the saved FM. The FM may have some features removed or their name modified but these changes are not taken into account. Therefore, XToF will warn the developer if existing taggings are no longer associated with an existing feature, but if the name of the feature is re-used by a new feature (the use of the path should restrict this possibility), the tagging will be considered as correct by the tool. Changing the associated FM is done at the risk of the developer.
- To compute the associated portion of source code, XToF uses the AST provided by Eclipse framework. XToF relies entirely on this. It cannot ensure that the same AST would be provided by other tools using the same approach and a one hundred percent identical association.

---

[1] A Java Applet enables testing Shrimp online http://www.thechiselgroup.org/projects Package dependencies features a visualization similar to what is proposed to use.

9.4. Future Work

- When several features are tagged in one tagging, the relation between them is an *OR*. This means, that if at least one of the features is selected, then the portion of source code is kept in the pruned product. To provide an *AND* for several features, the only possibility is to use one tagging per feature. As long as the same portion of source code is associated, the portion will be kept only if each tagging contains at least one feature selected. The tagging approach does not provide for a more complex condition. As there is not a negative option to prune when a feature is selected, there is no possibility of making any proposition. Research should be undertaken to provide more complex and complete conditions to prune a portion of source code. This would be useful when features have different codes that must be performed depending on the combination of features selected. For example, if feature A and feature B are present then a specific code must be kept as opposed to if only feature A or feature B was present.
- Documenting is an important activity in software development. As the tagging approach enables a tool to know how features are implemented, and as the tagging process is close to a JavaDoc process, the tagging information could be extracted to provide documentation on feature implementation. Research could be done on how this information should be extracted and presented and how useful it could be to the developers.
- One missing functionality in XToF is the creation and modification of FM. If an FM can be modified, the tool should provide propagation mechanisms to the tagging. Research should be carried out to identify under which conditions a modification could be accepted and propagated to prevent loss of tagging.

10. Conclusion

Software product lines proposes to automatically generate several softwares products according to different specifications. To achieve this goal, it must use features as a way to choose whether function are to be included or not. As software product line engineering adds processes to the development, it is important to reduce the unnecessary weight. The goal of the work was to search for a light weight approach to implementing the software product line.

There are two methods of implementing software product lines: the compositional and the annotative. As the first implies modification of the languages used and modification of the solution design it has a larger weight than the annotative approach that uses annotations to designate features without separating them. After comparing both approaches, two conclusions were drawn, the annotative approach is a lighter approach and most of its disadvantages can be compensated with a tool support.

Once the annotative approach was selected, a comparison of the different annotation methods was made. While some techniques were not adapted to software product lines, two were suitable. The first being CIDE, colouring features in the source code, and the tagging approach, using tags to annotate features.

Then the two methods were compared. For several reasons including the fact that CIDE uses a specific IDE to save and load annotations in contrast to the tagging approach that can be done manually, the second one was selected.

The tagging approach was then described in more detail and a tool support was specified as requested when comparing with the compositional approach. Using these requirements, a tool was built, XToF. First its architecture, backends and principles were described. The support brought by XToF aims at four processes, the tagging, configuring, pruning and finally, program understanding.

XToF provides different functions to support the developers in using the tagging approach to implement software product lines. As tests were done, they showed the need to improve the tagging support to achieve program understanding in the context of feature tagging, through the visualization.

In the background part, different visualizations were studied to help in software product line and program understanding. They helped develop two visualizations to support feature implementation understanding. Concern View emerged as the most adequate, by displaying a coloured matrix of numbers of taggings per file and feature. A third design, using class diagrams, was then compared.

Finally, two examples were described and helped extract lessons from the approach and from XToF. They presented in a practical context, the approach and the benefit of XToF to the developer to assist him in implementing and understanding feature implementation using tags.

The tagging approach and the support of XToF to the tagging approach (excluding the feature implementation understanding) were published in a tool demonstration paper at VaMOS [12]. The tagging approach supported by XToF can help developers implement software product line in a light weight.

Bibliography

1 Ball, T. and S. Eick, *Software Visualization in the Large.* IEEE computer, 1996. **29**(4): p. 33-43.

2 Benavides, D., P. Trinidad, and A. Ruiz-CortÈs. *Automated Reasoning on Feature Models*. 2005: Springer.

3 Boucher, Q., et al., *Tag and Prune : A Pragmatic Approach to Software Product Line Implementation*, in *Technical Report*, P.C. Research, Editor. 2009, University of Namur.

4 Cachopo, J. *Separation of Concerns through Semantic Annotations*. 2002: ACM.

5 Clements, P. and L. Northrop, *Software Product Lines.*

6 Coplien, J., D. Hoffman, and D. Weiss, *Commonality and Variability in Software Engineering.* Software, IEEE, 1998. **15**(6): p. 37-45.

7 Czarnecki, K., *Overview of Generative Software Development.* Lecture Notes in Computer Science, 2005. **3566**: p. 326.

8 Czarnecki, K., et al., *Generative Programming and Active Libraries (Extended Abstract).* 1998.

9 Czarnecki, K., S. Helsen, and U. Eisenecker, *Staged Configuration through Specialization and Multilevel Configuration of Feature Models.* Software Process: Improvement and Practice, 2005. **10**(2): p. 143-169.

10 Czarnecki, K., K. ÿsterbye, and M. Vˆlter. *Generative Programming*: Springer.

11 Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare, eds. *Structured Programming*. 1972, Academic Press Ltd. 234.

12 Gauthier, C., et al., *Xtofña Tool for Tag-Based Product Line Implementation.*

13 Heidenreich, F., I. Savga, and C. Wende. *On Controlled Visualisations in Software Product Line Engineering*. 2008.

14 Heymans, P., et al., *Requirements Engineering for Software Product Lines with Feature Diagrams*. 2008.

15 Horwitz, S., *Identifying the Semantic and Textual Differences between Two Versions of a Program*, in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, ACM: White Plains, New York, United States.

16 Kaelbling, M., *Programming Languages Should Not Have Comment Statements.* ACM Sigplan Notices, 1988. **23**(10): p. 60.

17 Kang, K., et al., *Feature-Oriented Domain Analysis (Foda) Feasibility Study*. 1990, Citeseer.

18 Kastner, C. and S. Apel. *Integrating Compositional and Annotative Approaches for Product Line Engineering*. 2008: Citeseer.

19 Kastner, C. and S. Apel. *Type-Checking Software Product Lines-a Formal Approach*. 2008.

20 Kastner, C., S. Apel, and M. Kuhlemann. *Granularity in Software Product Lines*. 2008: ACM New York, NY, USA.

21 Kastner, C., S. Trujillo, and S. Apel. *Visualizing Software Product Line Variabilities in Source Code*. 2008.

22 Kramer, D., *Api Documentation from Source Code Comments: A Case Study of Javadoc*, in *Proceedings of the 17th annual international conference on Computer documentation*. 1999, ACM: New Orleans, Louisiana, United States.

23 Krueger, C., *Easing the Transition to Software Mass Customization.* Lecture Notes in Computer Science, 2002: p. 282-293.

24 Krueger, C., *Software Mass Customization.* BigLever Software, Inc, 2005.

25 Leslie, D.M., *Using Javadoc and Xml to Produce Api Reference Documentation*, in *Proceedings of the 20th annual international conference on Computer documentation*. 2002, ACM: Toronto, Ontario, Canada.

26 Loughran, N. and A. Rashid. *Supporting Evolution in Software Using Frame Technology and Aspect Orientation*. 2003: Citeseer.

27 Mendonca, M., *Efficient Reasoning Techniques for Large Scale Feature Models.* 2009.

28 Mendonca, M., M. Branco, and D. Cowan. *Splot: Software Product Lines Online Tools*. 2009: ACM.

29 Northrop, L., *Sei's Software Product Line Tenets.* IEEE software, 2002. **19**(4): p. 32-40.

30 Oracle and Corporation. *Javadoc Tool Homepage*. Available from: http://java.sun.com/j2se/javadoc/.

31 Pawlak, R., *Spoon: Compile-Time Annotation Processing for Middleware.* IEEE Distributed Systems Online, 2006. **7**(11): p. 1-1.

32 Pohl, K., G. Bˆckle, and F. Van Der Linden, *Software Product Line Engineering: Foundations, Principles, and Techniques*. 2005: Springer-Verlag New York Inc.

33 Rubbani, H.H., *Semantic Web Solutions*, in *IT*. 2007, IT - University of Copenhagen.

34 Ryall, J., *Reminding and Refinding: Examining How Software Developers Use Annotations*. 2008, University of Victoria.

35 Schobbens, P., et al., *Generic Semantics of Feature Diagrams.* Computer Networks, 2007. **51**(2): p. 456-479.

36 Spencer, H. and G. Collyer. \# *Ifdef Considered Harmful or Portability Experience with {C} News*. 1992: Citeseer.

37 Stasko, J., et al., *Software Visualization*. 1998: Citeseer.

38 Storey, M., *Theories, Tools and Research Methods in Program Comprehension: Past, Present and Future.* Software Quality Journal, 2006. **14**(3): p. 187-208.

39 Storey, M., et al. *Shrimp Views: An Interactive Environment for Information Visualization and Navigation*. 2002: ACM.

40 Storey, M., et al. *Shared Waypoints and Social Tagging to Support Collaboration in Software Development*. 2006: ACM.

41 Storey, M., et al. *How Programmers Can Turn Comments into Waypoints for Code Navigation*. 2007: Citeseer.

42 Treude, C. and M. Storey. *Concernlines: A Timeline View of Co-Occurring Concerns*. 2009: IEEE Computer Society Washington, DC, USA.

43 Treude, C. and M. Storey. *How Tagging Helps Bridge the Gap between Social and Technical Aspects in Software Development*. 2009: IEEE Computer Society Washington, DC, USA.

44 Van Gurp, J., J. Bosch, and M. Svahnberg. *On the Notion of Variability in Software Product Lines*. 2001: IEEE Computer Society Washington, DC, USA.

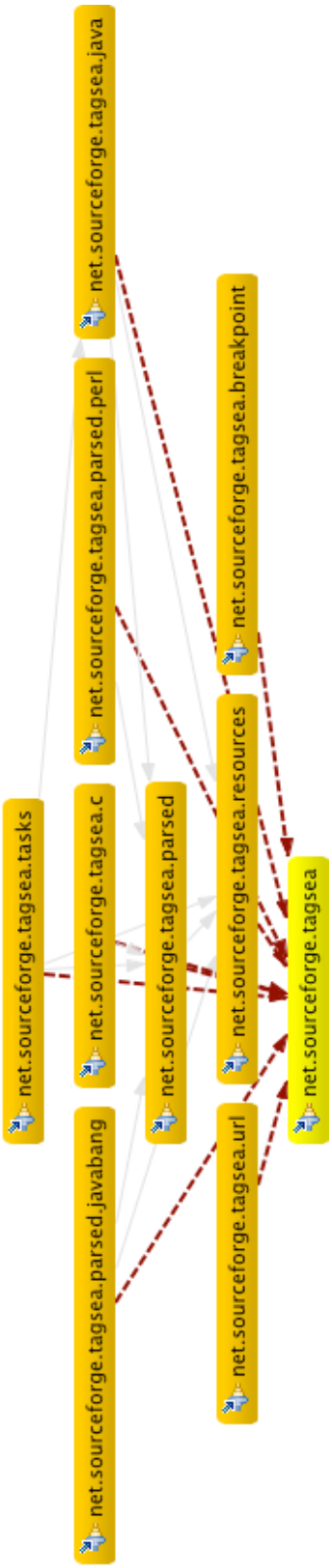
45 Wang, A. and K. Qian, *Component-Oriented Programming*. 2005: Wiley-Interscience.

46 Zhang, H. and S. Jarzabek, *Xvcl: A Mechanism for Handling Variants in Software Product Lines.* Science of Computer Programming, 2004. **53**(3): p. 381-407.

Appendix

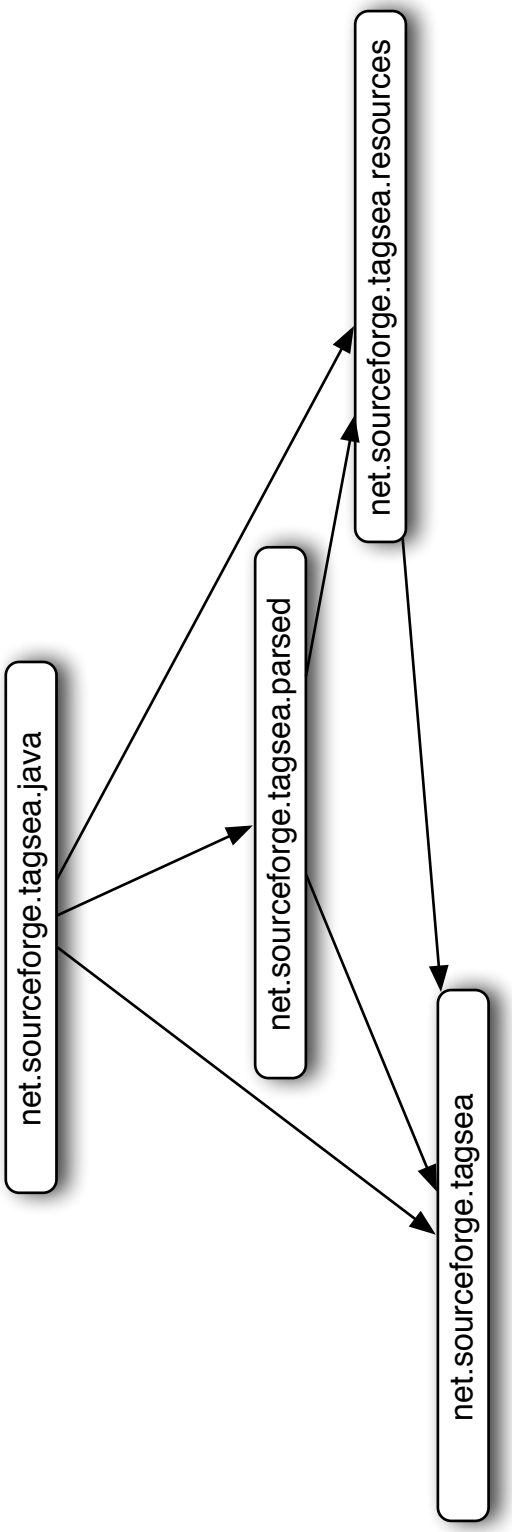
A. PDE Dependency View of TagSEA

Graph of dependency of plugins from TagSEA done with PDE dependency view tool.

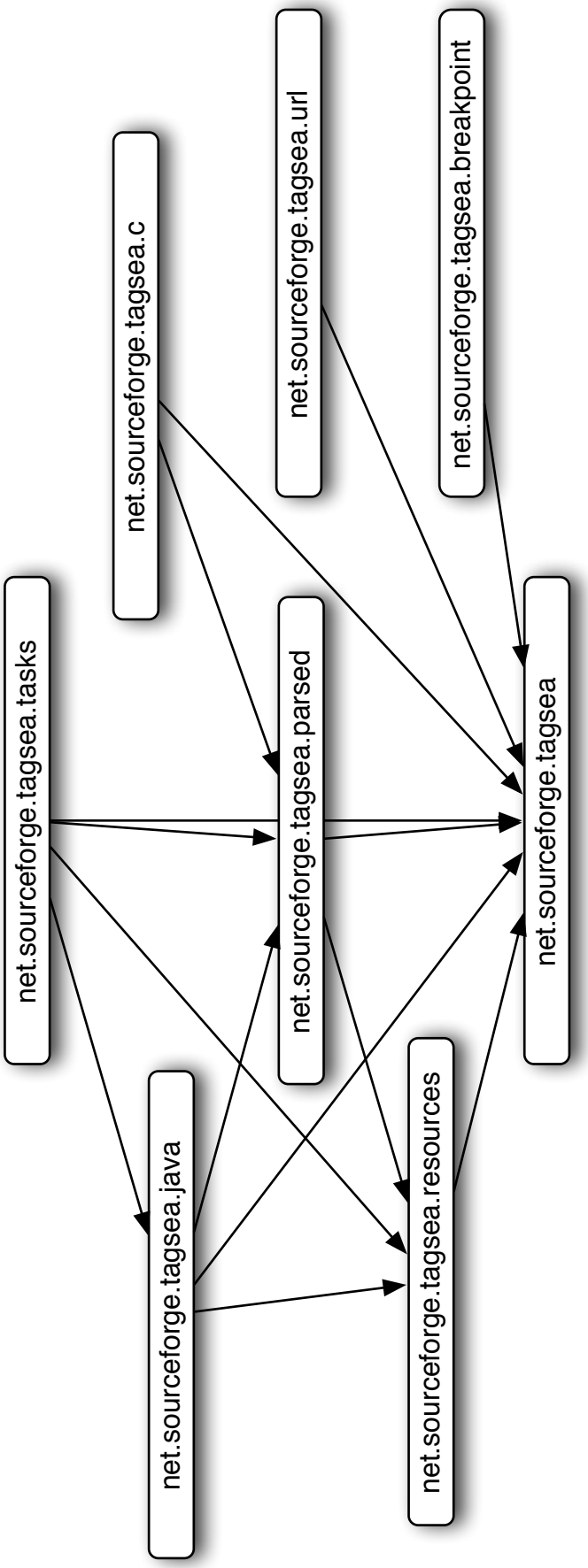


B. Dependencies TagSEA

Graph of plugin-dependencies of TagSEA. Core contains only plugin that are always installed. Extra also contain optional plugins.



Core


net.sourceforge.tagsea.tasks
net.sourceforge.tagsea.c
net.sourceforge.tagsea.url
net.sourceforge.tagsea.breakpoint
net.sourceforge.tagsea.parsed
net.sourceforge.tagsea
net.sourceforge.tagsea.java
net.sourceforge.tagsea.resources

Extra

C. SimpleEcho Feature Model

Source code of the feature model file from the SimpleEcho example

```
1.    <feature_model name="Simple Echo">
2.    <meta>
3.    <data name="description">A simple echo test</data>
4.    <data name="creator">Christophe Gauthier</data>
5.    <data name="address"/>
6.    <data name="email"/>
7.    <data name="phone"/>
8.    <data name="website"/>
9.    <data name="organization"/>
10.   <data name="department"/>
11.   <data name="date"/>
12.   <data name="reference"/>
13.   </meta>
14.   <feature_tree>
15.   :r Simple Echo(_r)
16.             :m display(_r_3)
17.                     :o display number words(_r_3_4)
18.                     :o display number of use(_r_3_5)
19.             :m additional features(_r_6)
20.                     :o keep history(_r_6_7)
21.                     :o count words(_r_6_8)
22.             :o command line(_r_9)
23.   </feature_tree>
24.   <constraints>
25.   constraint_1:~_r_3_4 or _r_6_8
26.   constraint_4:_r_6_7 or ~_r_9
27.   constraint_3:~_r_3_5 or _r_6_7
28.   constraint_5:~_r_3_4 or _r_9
29.   constraint_6:~_r_3_5 or _r_9
30.   constraint_2:~_r_6_8 or _r_6_7
31.   </constraints>
32.   </feature_model>
```

D. SimpleEcho Source Code

Source code of the files from the SimpleEcho Java project example.

I.Echo.java

```
1.    package simpleEchoSPL;
2.
3.    public class Echo {
4.
5.
6.            public String echo(String msg) {
7.                    String display = "I have been told that : "+msg;
8.                    System.out.println(display);
9.                    return display;
10.
11.           }
12.
13.           /*@feature:simple_echo.command_line@*/
14.           public String echoCommandResult(String msg) {
15.
16.                   String display = "Comand Result : "+msg;
17.                   System.out.println(display);
18.                   return display;
19.           }
20.
21.   }
```

II.EchoMain.java

```
1.    package simpleEchoSPL;
2.
3.    import java.io.BufferedReader;
4.    import java.io.IOException;
5.    import java.io.InputStreamReader;
6.
7.    /*@feature:simple_echo.command_line@*/
8.    import simpleEchoSPL.subPackage.CommandLine;
9.    import simpleEchoSPL.subPackage.History;
10.
11.   public class EchoMain {
12.
13.           /**
14.            * @param args
15.            */
16.
17.           public static void main(String[] args) {
18.                   Echo echo = new Echo();
19.                   /*@feature:simple_echo.additional_features.keep_history@*/

20.                   boolean hasHistory = true;
21.                   /*@feature:simple_echo.additional_features.keep_history@*/
22.                   History history = new History();
23.                   /*@feature:simple_echo.command_line@*/
24.                   CommandLine commandline = new CommandLine(history, echo);
25.
26.                   InputStreamReader iSR = new InputStreamReader(System.in);
27.                   BufferedReader reader = new BufferedReader(iSR);
28.                   String line;
29.                   try {
30.                           line = reader.readLine();
31.                           boolean echoDone = false;
32.                           while(!line.equals(""))
33.                           {
34.                                   String echoed = "";
35.                                   /*@feature:simple_echo.command_line@*/
36.                                   echoDone = commandline.executeCommand(line);
37.                                   if(!echoDone) {
38.                                           echoed = echo.echo(line);
39.                                   }
40.
      /*@feature:simple_echo.additional_features.keep_history@*/
41.                                   if(hasHistory && !echoDone)
42.                                           history.addEcho(echoed);
43.                                   line = reader.readLine();
44.                                   echoDone = false;
45.                           }
46.                   } catch (IOException e) {
47.                           e.printStackTrace();
48.                   }
49.
50.           }
51.
52.   }
```

III.subPackage/CommandLine.java

```
1.    package simpleEchoSPL.subPackage;
2.
3.    import simpleEchoSPL.Echo;
4.
5.    /**
6.     * A command is the format "command" ou "command i" where i is an integer
7.     * @author christophe
8.     *
9.     */
10.   /*@feature:simple_echo.command_line@*//*@!file!@*/
11.   public class CommandLine {
12.           private static final int CMD_UKNOWN = 0;
13.           private static final int REPEAT_LAST = 1;
14.           private static final int REPEAT_ID = 2;
15.           /*@feature:simple_echo.display.display_number_words@*/
16.           private static final int SHOW_WORD_COUNT = 3;
17.           /*@feature:simple_echo.display.display_number_of_use@*/
18.           private static final int SHOW_USE_COUNT = 4;
19.           private History history;
20.           private Echo echo;
21.
22.           public CommandLine(History history,Echo echo) {
23.                   assert(history!=null);
24.                   assert(echo!=null);
25.                   this.history = history;
26.                   this.echo = echo;
27.           }
28.
29.
30.           /**
31.            * Checks if the string is a command and then execute it
32.            * @param command The comand to execute
33.            * @return True iff the command has been executed
34.            */
35.           public boolean executeCommand(String command) {
36.                   assert(command!=null);
37.                   int commandId = getCommandId(command);
38.                   String msg;
39.                   switch(commandId) {
40.                   /*@feature:simple_echo.display.display_number_of_use@*/
41.                   case SHOW_USE_COUNT:
42.                           msg = "Use Count = "+history.getUseCount();
43.                           break;
44.                   /*@feature:simple_echo.display.display_number_words@*/
45.                   case SHOW_WORD_COUNT:
46.                           msg = "Word Count = "+history.getWordCount();
47.                           break;
48.                   case REPEAT_ID:
49.                           msg = history.getByID(getParamater(command));
50.                           break;
51.                   case REPEAT_LAST:
52.                           msg = history.getLast();
53.                           break;
54.                   default:
55.                           return false;
56.                   }
57.                   msg = echo.echoCommandResult(msg);
58.                   history.addEcho(msg);
59.                   return true;
60.           }
61.
62.           //Must be a valid command (getCommandId<>0)
63.           private int getParamater(String command) {
64.                   int index = command.indexOf(" ");
```

```
65.                     if(index<command.length()) {
66.                             try {
67.                                     return
    Integer.parseInt(command.substring(index+1));
68.                             } catch (NumberFormatException e) {
69.                                     return -1;
70.                             }
71.                     } else
72.                             return -1;
73.             }
74.
75.
76.             private int getCommandId(String command) {
77.                     boolean matches = command.matches("([a-zA-Z]+)|([a-zA-Z]+ [0-
    9]+)");
78.                     if(matches) {
79.                             String[] cmd = command.split(" ");
80.                             if(cmd.length>0) {
81.                                     String cmdMatch = cmd[0];
82.                                     if(cmdMatch.equalsIgnoreCase("RL"))
83.                                             return REPEAT_LAST;
84.                                     else if (cmdMatch.equalsIgnoreCase("RI"))
85.                                             return REPEAT_ID;
86.
    /*@feature:simple_echo.display.display_number_words@*/
87.                                     if (cmdMatch.equalsIgnoreCase("SW")) {
88.                                             return SHOW_WORD_COUNT;
89.                                     }
90.
    /*@feature:simple_echo.display.display_number_of_use@*/
91.                                     if (cmdMatch.equalsIgnoreCase("SU"))
92.                                             return SHOW_USE_COUNT;
93.                             }
94.                     }
95.                     return CMD_UKNOWN;
96.             }
97.
98.     }
```

IV.subPackage/History.java

```
1.    package simpleEchoSPL.subPackage;
2.
3.    import java.util.NoSuchElementException;
4.    import java.util.Vector;
5.    /*@feature:simple_echo.additional_features.keep_history@*//*@!file!@*/
6.    public class History {
7.
8.            private Vector<String> history;
9.            /*@feature:simple_echo.additional_features.count_words@*/
10.           private int word_count;
11.
12.
13.           public History() {
14.                   history = new Vector<String>();
15.           }
16.
17.           /*@feature:simple_echo.additional_features.count_words@*/
18.           public void computeWordsCount(String echo) {
19.                   String[] words_msg = echo.split(" ");
20.                   word_count = words_msg.length + word_count;
21.           }
22.
23.
24.           public void addEcho(String echo) {
25.                   /*@feature:simple_echo.additional_features.count_words@*/
26.                   computeWordsCount(echo);
27.                   history.add(echo);
28.           }
29.
30.           /*@feature:simple_echo.command_line@*/
31.           public String getLast() {
32.                   try {
33.                           return history.lastElement();
34.                   } catch (NoSuchElementException e) {
35.                           return "No Such element";
36.                   }
37.           }
38.
39.           public String getByID(int paramater) {
40.                   try {
41.                           return history.get(paramater);
42.                   } catch(ArrayIndexOutOfBoundsException e) {
43.                           return "No Such Index";
44.                   }
45.           }
46.
47.           /*@feature:simple_echo.additional_features.count_words@*/
48.           public int getWordCount() {
49.                   return word_count;
50.           }
51.
52.
53.           public int getUseCount() {
54.                   return history.size();
55.           }
56.
57.   }
```

E. Requirements of Tool Support

This table list all the requirements for the tool support and for each the associated priority. *R1* to *R14* are used in the work to reference them.

R	Function\Priority	High	Medium	Low
1	Load an FM	X		
2	Display an FD	X		
3	Create an FM			X
4	Checks Tags	X		
5	Auto-completion for Tags		X	
6	Drag-and-drop to tag			X
7	Display scope	X		
8	Display list of tagging location			
9	Navigate		X	
10	Configuration	X		
11	Retain state of a configuration		X	
12	Prune	X		
13	Minimal set of features		X	
14	Project Level visualization		X	